EXPLORING FORTH

OWEN BISHOP

Exploring FORTH

Exploring FORTH

Owen Bishop
in collaboration with
Audrey Bishop

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing 1984

British Library Cataloguing in Publication Data
Bishop, O. N.
Exploring FORTH
1. FORTH (Computer program language)
I. Title II. Bishop Audrey
001.64'24 Q.A.76

ISBN 0-246-12188-2

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

Contents

Using This Book

This is a book for those who are starting to learn FORTH. It does not expect any previous knowledge of the language. The explanations are detailed so that you can follow exactly what is going on at every stage.

The book will not teach you all there is to know about FORTH, as there is not enough room in a book of this size to deal with every aspect of this fascinating language. But it is hoped that, by the time you have finished this book, you will feel confident to continue the exploration of FORTH on your own, or with the help of one of the more advanced books.

This is not a book for armchair reading. You will get far more out of it and progress more rapidly if you have your computer switched on, with FORTH in operation, as you study the book. Key in all the short examples that are given throughout the book and watch things happen on the screen. Try varying the examples; type in something slightly different. See if the effect is what you expect it to be. If it is not, try to work out why. This is the practical way of exploring FORTH.

Chapter One
Which FORTH?

There has been an escalating interest in FORTH among micro owners during the past few years. As a result of this, the FORTH language is being made available on an increasing number of popular microcomputers. There are tapes, disks, cartridges and special ROMs, all of which provide FORTH for those micros which normally operate in BASIC. There is even a micro which has FORTH as its resident language. This book is intended to be used with any of these microcomputers, whatever version of FORTH they use.

The reason that this is possible is that FORTH is an easily transportable language. That is to say, you can write a 'program' on one micro, then key it into a different micro with a reasonable chance that it will work first time. The word 'program' was put into quotes in the previous sentence because the idea of 'writing a program' does not apply to FORTH as it does to BASIC and many other languages. As will be explained in more detail later, FORTH is based on a set of words, each of which has a specified action. The writer of a version of FORTH supplies you with a set of a few hundred words. When you 'program' in FORTH, you use these words to define new words of your own. You extend the language originally supplied to you by adding whatever words you need. You can then use the words you have defined to define even more words. The action of some of your words may be most elaborate. Yet everything is done in short, easily understood steps.

With BASIC and many other languages, you put together the statements and functions that are provided by the version of the language, building up line upon line of program. The program consists of a series of instructions telling the computer what to do. If the BASIC of your micro lacks certain statements which you need, you can often write a program line to do what is wanted, though sometimes this is difficult and it is always less satisfactory. For

example, if your BASIC lacks the REPEAT...UNTIL statements, you can manage with GOTO, but the program runs much more slowly.

There is nothing in FORTH which corresponds exactly to a program. It is true that there are the sequences of words used in defining other words, but these are short sequences more like subroutines or procedures than programs. On the other hand, FORTH words differ from subroutines or procedures because there is no 'main program' to jump back to after they have been executed.

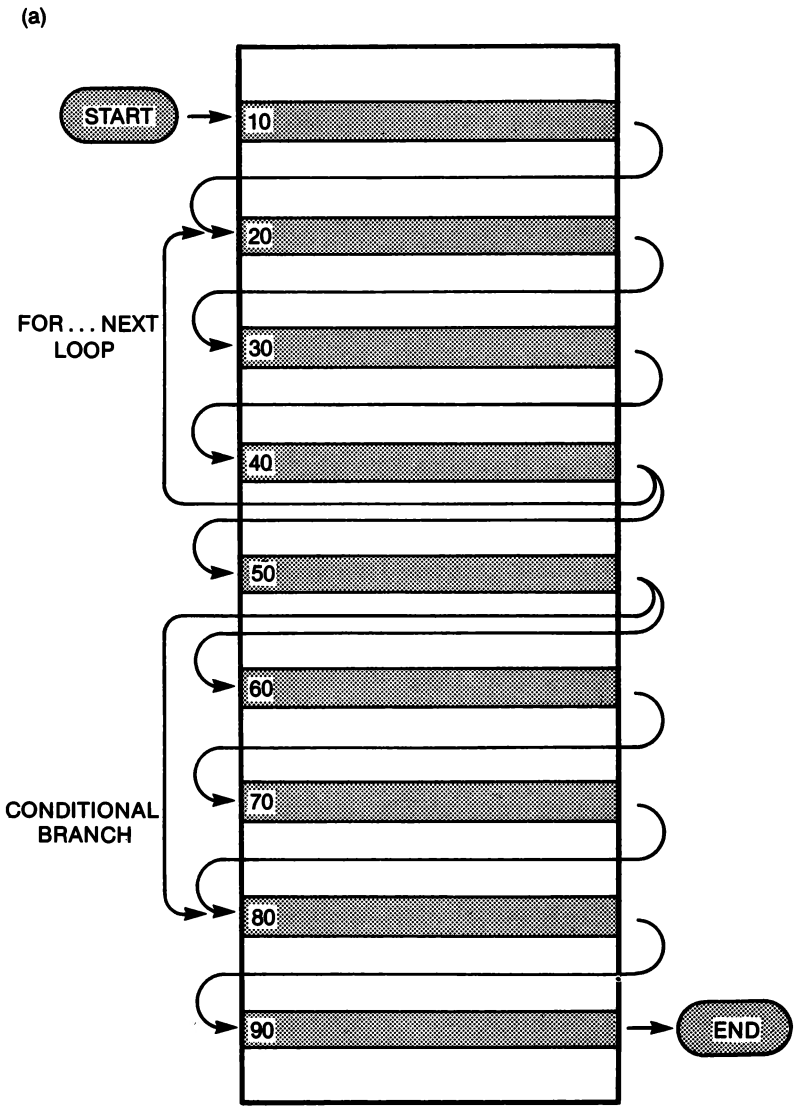


Fig. 1.1. The course of action when running (a) a program in BASIC; (b) an application in FORTH.

Figure 1.1 illustrates the difference between FORTH and a program-based language, such as BASIC. You can see how the action of a program proceeds line-by-line, perhaps with loops and repetitions, from the start of the program to the end. It is a *program* in the true sense; a list of things to be done.

FORTH has no such list. The action moves from one word to another. It threads its way among the words of the language itself. Exactly how this happens is explained later. FORTH is described as a *threaded language*. One word calls upon the actions of others in

(b)

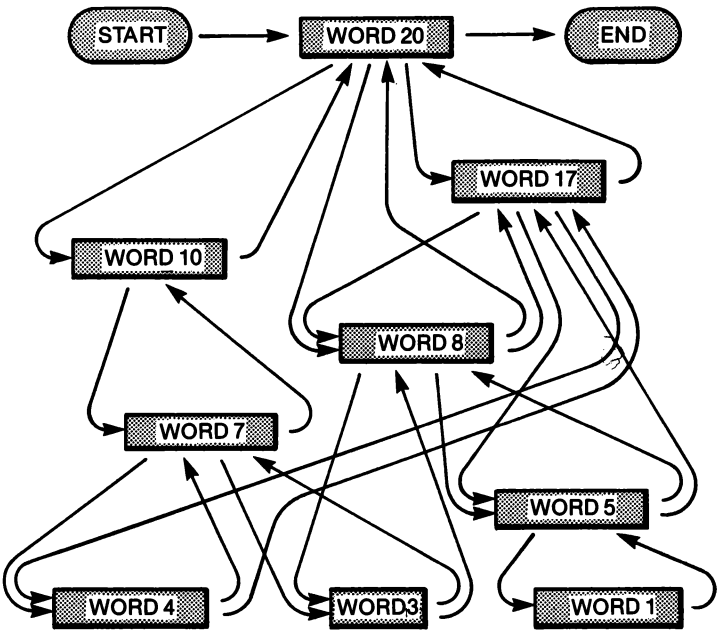


Fig.1.1 (contd)

performing its own actions. There is no 'listing' that you can read from start to finish, to see what the program does. To follow the action you must thread your way from word to word. But, since the definition of each word is short and clearly related to the definitions of other words, this is an easy matter. When working with FORTH, we use or apply the existing words to create new words. For this reason, it is best to speak of an *application* rather than a 'program'.

We will return to this topic in more detail in various parts of the book.

Most of the words that are provided with your version of FORTH will be the same as those provided in other versions and will have the same action. There may be a few additional words which the designer thinks you will find helpful. There will also be words which relate to the special features of your computer and are not present in other versions. For example, the Acornsoft FORTH for the BBC Microcomputer has a word MODE which is used for changing the display mode of the computer. This word does not apply to other computers, such as the Jupiter Ace. But the FORTH of the Jupiter Ace has some words of its own, including BEEP, to make a beeping sound on its loudspeaker. The BBC FORTH does not have BEEP, though you could write it using BBC FORTH if you wished to. As far as possible, the words used in this book will be only those which are likely to be found in all versions of FORTH.

What if this book uses a word which is not in your FORTH? This does not happen often, but there are some generally useful words which may not be in your version. Here we can rely on the ability of FORTH to let you define the missing word yourself from the words you have already. Some definitions based on the essential FORTH words are given in Appendix A.

There are two main versions FORTH. One of these, FORTH-79, is defined by a standard set out by the FORTH Standards Team. The other main FORTH is that prepared by the FORTH Interest Group in the United States of America. It is called fig-FORTH. These two versions have a lot in common. As far as possible, this book uses words which occur in both versions. Where there are any significant differences, we shall try to point them out. Although FORTH-79 is a standard, there are many versions of it. The point is that the standard specifies a minimum set of words and how they are to act. Any FORTH which has this minimum set and which has a number of other standard features can claim to be FORTH-79. The writers of such a version are then free to add any other words, especially words like those mentioned above which cater for the features of a given micro. Provided that special words are avoided, an application can generally be transferred from one micro to another, without any problems. On the other hand, the words which apply to the special features of a micro are usually those which provide the most effective displays or make best use of features such as sound generators. Applications which do not make use of such words are not exploiting the features of the micro to the best

advantage. This dilemma is not the fault of FORTH, but of lack of standardisation in micros, especially with regard to screen format and display routines. This is the point at which you may need to adapt the suggestions given in this book to suit the special features of your computer. Guidance is given wherever possible. By the time you have covered the early chapters of this book, you will feel confident to use the special words of your version of FORTH or to write any other words you need to get the most out of your computer.

To summarise

In this chapter you have learned that:

- Writing applications in FORTH consists of using FORTH words to define new FORTH words of your own.
- FORTH is a flexible, transportable language.
- There are two main versions of the language, FORTH-79 and fig-FORTH.

Chapter Two
Why FORTH?

The increasing popularity of FORTH is due to several factors. One of these, already mentioned, is its *transportability*. This becomes an increasingly important factor as more varieties of microcomputer come on the market. Another factor is its *speed*.

FORTH is fast in two ways. It is claimed that writing an application in FORTH takes only half the time required to write the equivalent program in another high-level language, such as BASIC. A FORTH application takes only one tenth of the time required to write its equivalent in assembler. So here is a way to get your computer into action with the minimum of delay at the writing stage.

Once the application is written, it runs faster too. To check on this, let us see how long it takes the micro to count up to 30000, using BASIC. Here is the program which does it. This took 16 seconds to run on the BBC Microcomputer.

```
>10 FOR J = 1 TO 30000
>20 NEXT
```

Now run FORTH on your computer and key in the following word definition. Type this in exactly as shown below, taking special care to leave all the spaces. FORTH is particular about spaces!

```
: TEST 30000 1 DO LOOP ;
```

When you have finished, press RETURN. The computer responds with the familiar and reassuring 'OK' on the line below. This tells you that it is ready for whatever you want it to do next.

Before doing anything further, consider what you have just typed. The line defines a word named TEST. It was given this name as we want to use it to test how fast the computer counts when using FORTH. The name of the word is followed by two numbers. Comparing this line with the BASIC listing above, shows that these are the values for the end and the beginning of the loop. You will

notice that the number for the end of the loop comes *before* the number for the beginning. This 'back-to-front' habit of FORTH is something we shall see a lot more of. Why it works this way is explained later. It may seem strange at first, but you soon get used to it, just as one soon gets used to driving a car on the opposite side of the road when visiting a foreign country. Then it feels odd when you come back home!

The loop which does the counting begins with the word DO and ends with the word LOOP. In this definition there is nothing between DO and LOOP, just as there was nothing in the BASIC loop given earlier. All the micro is being asked to do is to loop back to DO thirty thousand times.

Now to execute the word TEST. Key in the word TEST by itself on the line below. Get your stop-watch ready, then press RETURN. The 'OK' message appears as soon as the computer has counted to 30000. On the BBC Microcomputer, the test took only two seconds. For this particular operation, FORTH is about eight times faster than BASIC. Comparisons for other operations give different results, depending on what has to be done by the micro, but it seems that the claim that FORTH is up to ten times faster than BASIC is to be believed.

The speed of FORTH makes it ideal for computer games. It also has the advantage that the computer can perform a long series of calculations in a reasonably short time. There are applications in this book which take advantage of the speed of FORTH in both of these ways.

High-level languages are of two main types: *interpreted* and *compiled.* BASIC is an interpreted language. When a program in BASIC is run, the computer goes through it, line by line, working out what the various statements mean. The interpreter program in the ROM of the micro interprets each BASIC statement, calling on machine code routines to perform the necessary actions. The program is interpreted every time it is run. Moreover, if there are lines in a loop which are repeated, say 100 times, then those lines have to be interpreted 100 times each. Interpreting inevitably requires time, which means that BASIC programs run relatively slowly. But interpreters have an advantage. When you are working with an interpreted language, it is easy to stop the program, make small changes in it and then run it again. Programming is relatively quick and easy, and you can instantly see the results of any changes you make.

A compiled language, such as Pascal, has entirely different

features. After you have written the *whole* program, it is compiled into a machine code version which can be stored on tape or disk. After that you use the machine code version. The conversion of the program from a high-level language to machine code is done once and for all. The compiled version of your program runs exceedingly fast, which is a big advantage. The corresponding disadvantage is that it is not possible to make any changes in the program without starting from the beginning and recompiling the new version. This is annoying if there are bugs in the program, as there are almost certain to be at first. It is much more important to get the program right before it is compiled. Some people would say that this is a good thing, for it forces you to work carefully and plan everything in detail before you begin to program. Needless to say, a lot of other people are put off by this strict approach to programming, which is perhaps one reason why Pascal has never become popular with micro owners. Another disadvantage of compiled languages is that they generally require more memory than is present in the average micro.

FORTH is neither interpreted or compiled, in the usual sense of these words. When you *define* a FORTH word, the name of the word is stored away, together with a string of numbers which link your word to words you have used in the definition. Once a word has been defined, it becomes a part of the language. This is roughly equivalent to compiling, but it does not produce machine code. It produces a word which refers to other words already compiled. Some of these words, the *primitives*, perform the more fundamental actions. The primitives have links to routines in machine code which, in effect, will do all the real work when the word is executed.

When a word is *executed*, during the running of an application, FORTH acts more like an interpreter. As each word is executed, several other words may be called into action. Some of these call on primitives which in turn bring various machine code routines into operation. Since the words have already been 'compiled', interpretation is very fast. This gives FORTH its speed. However, since the words of FORTH are each individually 'compiled', we have an extremely flexible and accessible system. You define words one at a time, and can test each word thoroughly before you go on to write the next. If, later, a definition turns out to be unsatisfactory in some way, you can re-define it without having to re-compile the rest of the application. In this and other ways, FORTH allows the user to interact freely with it, one of the advantages of an interpreted language, yet has the speed and compactness of a compiled

language. It combines the advantages of both.

Since the core of FORTH has a relatively limited number of tasks to perform, it is short and requires little memory. The core routines, and essential word definitions of a typical implementation of FORTH require only about 8 kilobytes of memory. In any given application you need add only those words which are required by the application. This keeps memory requirements to a minimum.

The flexibility of FORTH has been mentioned already. The language is a highly structured one, yet you are allowed the freedom to alter its structure to suit your own purposes. We shall see examples of this in later chapters. It is considered by some that a language that can alter itself and extend itself so readily is not really a language at all! This flexibility gives it several advantages in the field of education. Other languages present the user with a rigid system of statements or commands, and fixed ways of doing things. This implies that the user has quite a lot to learn before beginning to use the language. If there are difficult aspects of learning the language, it is not possible to alter the language to make it easier. With FORTH we can make things much easier for the beginner or those with special difficulties. Words can be defined which do very simple and easily understood things. We can give them names which readily make sense to the learner. Words from the user's everyday vocabulary are more likely to convey meaning than words chosen by someone else. The beginner can give any preferred name to the words. One can go further. If the name of an existing FORTH word is confusing to the user, there is no difficulty in re-defining it with an entirely different name. For example, 'duplicate' is a word in English not readily understood by young people, so the FORTH word DUP may not be understood either. Perhaps MAKE-TWO would be better understood in the earlier stages of learning FORTH. To add this to the language, all that is required is:

```
: MAKE-TWO DUP ;
OK
```

From then on MAKE-TWO will have exactly the same action as DUP. In the same way it is easy to adapt the FORTH vocabulary to suit those whose mother tongue is not English.

It must be evident from the above that the writers have a high opinion of FORTH and its potentialities. Turn now to Chapter Three and begin to experience the delights of FORTH for yourself.

To summarise

In this chapter you have found out how to:

- Key in and use FORTH words.
- Define a new FORTH word, using existing FORTH words.
- Use a DO ... LOOP loop.

You have learned that:

- FORTH is fast.
- FORTH combines the advantages of an interpreted language with those of a compiled language.

Chapter Three
Stacking It Up

The stack is at the heart of all operations in FORTH. It is therefore very important to understand what the stack is and how it is used.

The stack is a part of memory specially set aside for holding numbers. It is not a very large part of memory, usually less than 100 bytes. The word 'stack' implies rather more than the related word 'heap'. In a stack, things are arranged in some kind of *order*. There are several ways of thinking about the stack. One of these is to imagine a stack of postcards in a clip-board (Fig. 3.1). Putting them on the clip-board stops the cards from getting out of order. In other words, it prevents the stack from turning into a heap!

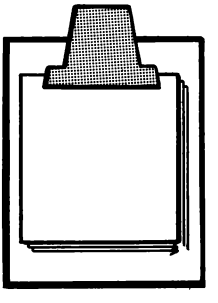


Fig. 3.1. The clip-board ready to demonstrate the stack.

When we look at the stack, we see only the top card. We will refer to this as *top-of-stack*. This particular card is only top-of-stack for as long as the stack remains unaltered. We could add another card to the stack, placing it on top of the stack. Now the original top-of-stack card is covered and the newly added card becomes the top-of-stack. Or we could remove the top-of-stack card and throw it away. Then the newly exposed card which was below it becomes the new top-of-stack.

Note that when we add a card to the stack we always place it on

top. We never try to insert it further down in the stack. Also, when we remove a card from the stack, we always remove the *top* card, never a card from further down the stack. These two rules are an essential part of the way the stack works.

Suppose we begin with an empty clip-board. This can be referred to as an 'empty stack'. It would help at this stage if you were to have an empty clip-board beside the computer. You also need about ten cards (or sheets of paper), and a pencil. If you do not have a clip-board it does not matter; just keep the windows and doors shut so there is no wind to blow the stack away! When you are ready with this equipment, turn on the computer and call up FORTH.

First write the figure 4 on one of the cards (we will refer to them as cards, even if you are really using scraps of paper). Place it in the clip-board. This card is top-of-stack. The value stored at top-of-stack is 4 (Fig. 3.2).

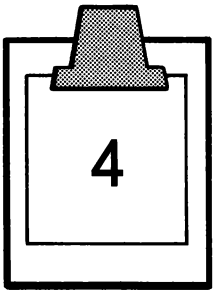


Fig. 3.2. Value 4 at top-of-stack.

When you first run FORTH, its stack is empty, just like the clip-board was. To place 4 at top-of-stack, all you have to do is type:

```
4
```

and press RETURN. You will see the 'OK' prompt on the next screen line, indicating that your instructions have been obeyed and the computer is waiting to be told what to do next.

How do we know that 4 has really been placed at top-of-stack? FORTH has a word which tells the computer to take the top number off the stack and display it. This word is the shortest word possible:

```
.
```

No, it's not a dirty mark on the page, it is a full-stop. When we refer to it, we call it 'dot'. Key in 'dot', then press RETURN. The sequence so far is shown in Fig. 3.3. As you can see, the computer has displayed 4, the number stored at the top-of-stack.

```
4
OK
.
4 OK
```

Fig. 3.3.

Can you get the computer to display it again? Try 'dot' followed by RETURN. What happens next depends on the version of FORTH you are using. On the BBC Microcomputer, for example, you get:

```
.
0 .  ? MSG # 1
```

The words MSG # 1 refer you to error message no. 1. If you look this up in the manual, you will find that it means 'Stack empty'.

On the Jupiter Ace you will get:

```
-18572ERROR 2
```

The first number may be different, but the error message will always be number 2, which means 'Stack empty'.

It seems that the 4 is no longer on the stack. It is the same as if you had taken the card from the clip-board, and pinned it up on the wall for everyone to see (Fig. 3.4). Do this now – take the card from the board and put it where everyone can see it (no need to pin it to the wall!).

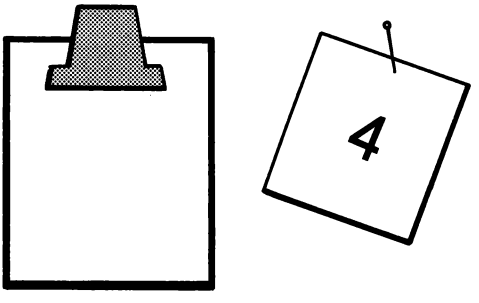


Fig. 3.4. Value 4 displayed, leaving the stack empty.

To sum up so far: we put 4 on the stack. Then we used 'dot'. The action of 'dot' was to take the number from the top-of-stack, leaving the stack empty, and display the number on the screen.

Now put the 4 card back on the board, so it is once more top-of-stack. Next write 55 on another card and place this on top of the 4 card. The 55 card is now top-of-stack. The 4 card has become second-on-stack. Finally write 666 on a third card and put this on top of the 55 card (Fig. 3.5). Remember we always add to the top of

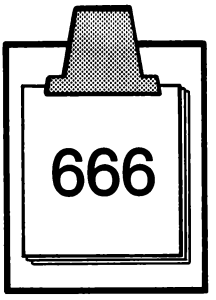


Fig. 3.5. Three values, 4, 55 and 66, on the stack.

the stack. There is no slipping the 666 card under the 55 card! We can do the same thing with the computer's stack (see Fig. 3.6). Press

```
4
OK
55
OK
666
OK
```

Fig. 3.6.

RETURN after keying in each number. Which number is at the top-of-stack now? Use 'dot' to find out. Then use it twice more. Figure 3.7 shows what happens. The numbers are taken off the stack and

```
. 
666  OK
. 
55  OK
. 
4  OK
```

Fig. 3.7.

displayed in the reverse order to that in which they were put on the stack. That this should happen is obvious if you take the cards from the top of your clip-board stack. First comes the 666. Removing this

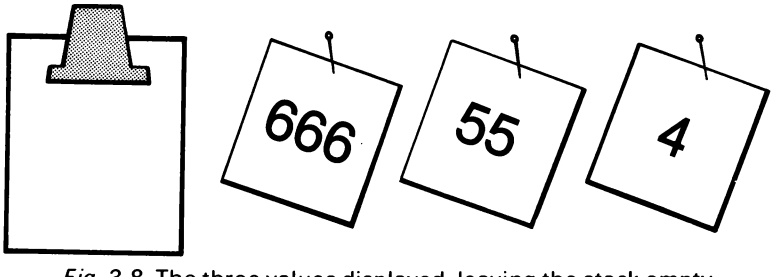


Fig. 3.8. The three values displayed, leaving the stack empty.

and propping it up on a shelf exposes the 55, the new top-of-stack. Take this away and prop it up on the right of the 666. This leaves the 4, which you remove and place to the right of the 55. Now the stack is empty once again (Fig. 3.8).

The stack is what is known as a last-in-first-out stack. This is often shortened to LIFO. The 666 was the last to be put on the stack, and it is the first to be taken off.

In the sequence above we pressed RETURN after keying in each number and pressed RETURN after each dot. The computer can accept and perform any reasonable number of instructions in one go. Key in the numbers and 'dots' all on one screen line:

```
4 55 666 . . .
```

Recall the earlier remarks about spaces; leave one space between each number and each 'dot'. Then press RETURN. The result is:

```
666 55 4 OK
```

Adding on the stack

If all that FORTH could do is stack numbers and then unstack them in reverse order, it would be a useless language. Let us give it more to do with the numbers. But first, try this with your clip-board stack. Begin with an empty stack. Place the 4 card on the stack, then the 55 card. Now comes the adding. Take both cards off the stack ready for adding. Place them and a blank card as shown in Fig. 3.9. Add 4 and 55 to get the answer 59 and write this on the blank card. Put the 59 card on the stack. Having done the addition, and having placed the answer on the stack, you have no further interest in the 4 and 55 cards. So throw them away. This is just what the micro does.

The same operation on the computer is done in three steps:

(1) key in the first number (4)
(2) key in the second number (55)
(3) tell the micro to add them together and place the result on the stack.

On the screen this looks like:

```
4 55 +
OK
```

The 'OK' shows that RETURN has been pressed and the job is done. What about the answer? Oh yes, that is at the top of the stack. The

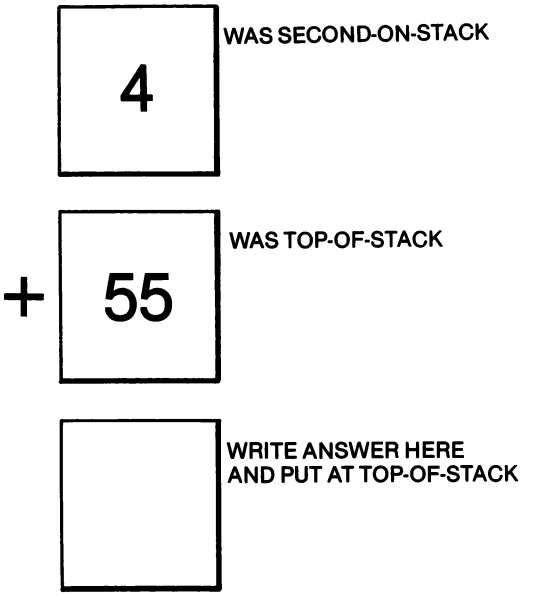


Fig. 3.9. Adding, with two values taken from the stack.

computer will not show you what the answer is unless you tell it to. Use 'dot' to find out the answer:

```
. 59  OK
```

This leaves the stack empty – or should do. Use 'dot' again to find out if the 4 or the 55 are still lurking on the stack. They are not there. They have been used as requested and are now forgotten.

There is one thing you may not have noticed about what you just did. It all happened so naturally that it did not seem strange. To make the micro add 4 and 55, you gave it the two numbers *first* and *then* told it to add them. This makes sense. Until the two numbers are on the stack, how can the computer be expected to deal with them?

Yet some people who are beginning FORTH find this a strange way of going about things. This is because we usually write down an addition as:

$$4 + 55 = 59$$

The + comes *between* the two numbers which are to be added. But this is only a convenient way of writing down the numbers as an equation. You do the actual addition in your head. When you were first learning to add, you probably wrote it down (or had it written

down for you) like this:

```
First one number              4
Then the other number       + 55
Then the '+' to tell          ___
you to add them together
```

This is just the way it is done in FORTH, using the stack. Now try this. Empty your clip-board stack. Put the 666 card on the stack, then put the 59 card on top of it. 59 is top-of-stack and 666 is second-on-stack. Now we are ready for a subtraction. Take the 59 off the stack, then take off the 666, placing it on the table above the 59 card. Put a blank card below the 59, as in Fig. 3.10. Do the subtraction, writing

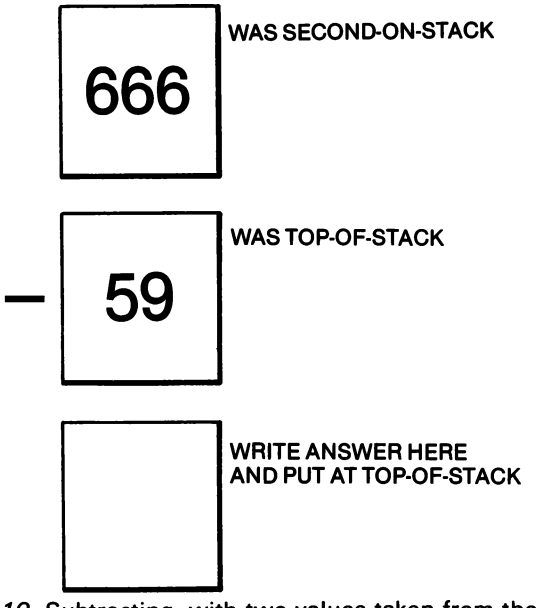


Fig. 3.10. Subtracting, with two values taken from the stack.

the answer (607) on the blank card. Place this card on the stack. The 59 and 666 cards are not needed any more, so throw them in the waste-paper basket. The subtraction is done; now to let other people see the answer. Take the 607 card from the stack and place it on a shelf.

Here is the same thing done in FORTH:

```
666 59 - .
607 OK
```

This does three things:

(1) Put one number (666) on the stack

(2) Put the other number (59) on the stack
(3) Tell the computer to subtract

We have included a fourth item, the 'dot', in the line, so we get the answer straight away, and nothing is left on the stack. As with the addition, we give the computer the two numbers *first*, and *then* tell it what to do with them. As before, we cannot expect it to do anything until it has the two numbers to work on. Numbers first, action second.

There is one way in which addition is different from subtraction. If we tell the computer to add, it does not matter in which order the two numbers have been placed on the stack. If we tell it to subtract, it always takes the top-of-stack from second-on-stack. To check on this, try:

```
59 666 - .
-607 OK
```

Taking a larger number from a smaller one gives the expected negative result.

Stack rules OK?

At first glance the heading above looks as if it is a line of FORTH, but it is not. It indicates a breathing point for summing up the rules by which the stack operates. The two rules about the top-of-stack can be put together as one rule:

The TP rule *T*ake and *P*ut at the *T*o*P* of stack

In other words, numbers taken from the stack are always taken from the top; new numbers put on the stack are always put on the top.

The second rule is about giving the micro the figure or figures to work on, and then telling it what to do with them. We can call this rule:

The FFAA rule *F*igures *F*irst, *A*ctions *A*fter

Readers who are soccer fans may prefer to call this the FA rule.

The TP rule explains why the DO...LOOP line given at the beginning of Chapter Two seemed to have the figures the wrong way round. Now it is clear that they are the right way round. The loop is to run from 1 to 30000. When the computer wants to know how many LOOPs it is to DO, it will go to the stack and look at top-of-stack. Here it will find 1, for this was the figure placed there last. It is

the first to come off (LIFO is a result of the TP rule). Having found out the value which is to begin the loop, it wants to know the value which is to finish it. The micro looks at the stack again. It has already taken away the 1, so now the top-of-stack is 30000. It removes this, leaving the stack empty, and performs the loops. This line also illustrates the FFAA rule. The figures for the end and the beginning of the loop are put on the stack, *then* the micro is told to DO a LOOP. More about DO...LOOP in Chapter Nine.

Words and numbers

In FORTH, anything you type on the screen is either a word or a number. Words and numbers are always separated from each other by at least one space. They may even be on separate lines. When the computer interprets a line of FORTH, it examines each group of characters on the line, working from left to right. If a group of characters consists entirely of numerals, it recognises the group as a number. If any one or more of the characters are letters or symbols, the group is a word. This means that words can consist of groups of letters:

```
DUP ROT DROP OVER LOOP
```

or letters and symbols:

```
C! ?TAB (+LOOP) M*
```

or symbols alone:

```
: + . ( ; */ -->
```

or letters and numerals:

```
2DROP 4HEX 79-STANDARD
```

(The word can *begin* with numerals if you wish, which is something not allowed in BASIC.) Spaces are not allowed in words, which is obvious, since FORTH relies on spaces to separate one word from the word next to it. If you want a 'two-word' word, use a hyphen as in the example MAKE-TWO at the end of Chapter Two.

Although it was stated above that only an 'all-numeral' group of characters is read as a number, there is the exception that a group of numerals with a minus sign in front is taken as a negative number. In some FORTHs you are allowed to key in numbers with decimal points – what are called *floating point* numbers. The FORTH–79

standard does not provide for such numbers. All its calculations are done in whole numbers, or *integers*. Using floating-point numbers makes calculations slower and there is not usually any real need to work with such numbers. We shall consider this topic again in Chapter Eleven.

The words we have used in Chapters Two and Three are:

```
. + - DO LOOP : ;
```

The action of 'dot', 'plus', 'minus', DO and LOOP have already been explained, but we have not mentioned 'colon' and 'semicolon'. The word 'colon' tells the computer that you are defining a new FORTH word. We used it at the beginning of the line in which we defined the words TEST and MAKE–TWO (Chapter Two). The word 'semicolon' is used to tell the computer when the definition of the word is complete.

Writing out the stack

Using clip-boards and pieces of paper is helpful in getting to know how the stack operates, but there are quicker and neater ways of doing things. From now on we shall use a way of indicating the content of the stack which is used in a number of other books on FORTH. It is a very useful way of showing what the stack holds, both before and after a given operation.

If we were to put 4 on the stack, then 55 and then 666 we would *write out* such a stack like this:

```
4\55\666
```

As you can see, top-of-stack is on the right. If we want to show what an operation does, we set it out like this:

```
(4\55 ... 59)
```

This shows what happens when we add 55 to 4. The way the stack begins is on the left. The way it finishes is on the right. The three full-stops (...) separate start from finish. Some books use a dash instead. This is what 'dot' does to the stack:

```
(32  ...)
```

This example shows that 'dot' begins with a number (e.g. 32) on the stack and finishes with nothing.

Quite often we want to show what a word does without using any numbers in particular. Then we use 'n', or 'n1', 'n2', 'n3', etc., if there is more than one number. The action of the word 'dot' can be shown by:

(n ...)

The action of the word 'plus' is shown by:

(n1\n2 ... n1 + n2)

You begin with n1 as second-on-stack and n2 as top-of-stack and you finish with their sum (n1+n2) as top-of-stack. The action of the word 'subtract' is:

(n1\n2 ... n1 − n2)

Top-of-stack is subtracted from second-on-stack.

It helps us follow the more complicated operations if the computer will write out the contents of the stack for us at various stages in the operation. We cannot simply use 'dot' repeated several times, because 'dot' takes every number off the stack as it displays it. The word for displaying the whole stack at any given time is 'dot-S'. Figure 3.11 shows it in action. We have used 'dot-S' after every stage in the operation, to show just what has happened to the stack. Using 'dot-S' has not altered the stack in any way.

```
4 .S 55 .S 666 .S + .S - .S . .S
4
4 55
4 55 666
4 721
-717 -717
EMPTY OK
```

Fig. 3.11.

To summarise

In this chapter you have found out how to:

- Use the stack.
- Find out what values are on the stack.
- Write out the stack contents systematically, on paper.
- Write out the stack action of FORTH words.

You have used these FORTH words (stack action of words in brackets):

- . 'dot' displays the value at the top-of-stack (n...).
- .S 'dot-S' displays the whole stack, without changing it.
- + 'plus' (n1\n2 ... n1+n2).
- − 'minus' (n1\n2 ...n1−n2).
- DO and LOOP are words used to begin and end an action that is to be repeated a given number of times (n1\n2 ...).
- : 'colon is used to begin a colon definition of a FORTH word (...).
- ; 'semi-colon' is used to end a colon definition (...).

You have learned that:

- OK on the screen means that the computer is waiting for your next instruction.
- Most FORTH operations depend on putting values on the stack or taking them off the stack.
  The two stack rules are:
  TP − *T*ake and *P*ut to *T*o*P* of stack
  FFAA − *F*igures *F*irst, *A*ction *A*fter

Explore more

(1) Key in lines to make the computer do the following additions and subtractions and display the result:

23 + 56	146 + 72	146 − 72
4 + 43 + 5	1 + 0 + 67	1000 + 100 + 10
45 + 21 − 6	55 − 3 − 2	80 − 2 + 22

(2) Define a word ADD to take three numbers from the stack, add them together and display the result.

(3) Define a word SUBTRACT to take three numbers from the stack and subtract the number at the top-of-stack from the sum of the other two.

Chapter Four

What Is The Stack?

It is a good idea to think of the stack as a pile of cards on a clipboard or as a row of numbers on the page or on the screen, but obviously it is not really like this. It will help you to understand the way the stack works and to learn how to use it more effectively if you find out a little more about it.

We have already said that the stack is a small section of the computer's memory. To be more specific, it is part of the Random Access Memory (RAM), and it can be written into and read from whenever we want. In most micros each location in memory consists of a single byte of eight bits. We can think of each location as a set of eight registers, each of which can hold either 0 or 1. Of course there is nothing there actually looking like a 0 or a 1, but there are electrical circuits, called *flip-flops*, which can be in one of two states. When flipped they can be said to represent a 1 and when flopped they represent a 0. Since only 1s and 0s can be represented in such a system, any number stored must be a binary number. That is to say, it is a number made up entirely of 1s and 0s. The computer can show us what binary numbers look like. Here is a word which converts ordinary decimal numbers into binary ones and displays the result:

```
: BINARY 2 BASE ! . DECIMAL ;
OK
```

Later we will see just how this word works but for the moment it is enough to just use it. Use it by typing any decimal number you like (remember – *F*igure *F*irst) then type the word (*A*ction *A*fter) – see Fig. 4.1.

If you type in numbers greater than 127, the binary number has eight digits, which is the maximum that can be stored in a single memory location. The largest possible number that can be stored is 255, which is all ones, 11111111. The word BINARY can work with numbers far larger than this, but these would need more that one

```
4 BINARY
100 OK

5 BINARY
101 OK

32 BINARY
100000 OK

345 BINARY
101011001 OK
```

Fig. 4.1.

byte to store them. This is a point we shall return to in a moment.

The computer knows where to go to find the stack because each location in its memory has an address. In most micros the address is a number between 0 and 65535. The manual which goes with your version of FORTH may tell you the address at which the stack is kept, though you need not worry if it does not. The computer knows and that is all that matters. Figure 4.2 shows a part of memory in which the stack values are stored. The addresses shown in the figure are probably quite different from those actually used in your computer. Why the address of each location in the stack differs from those of its neighbours by two is explained later.

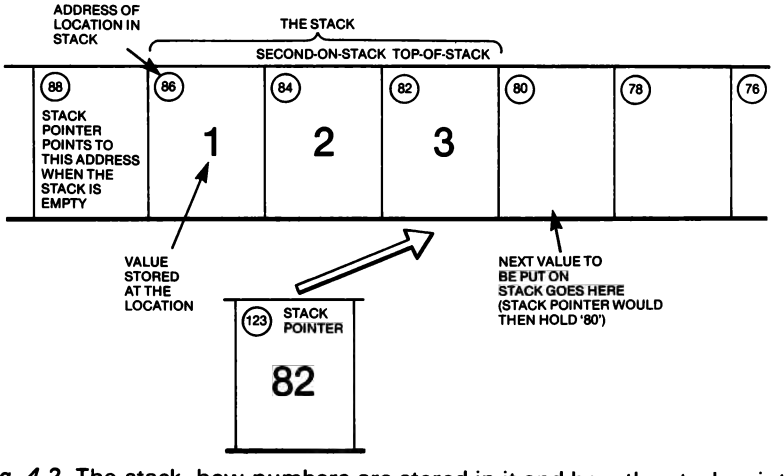


Fig. 4.2. The stack, how numbers are stored in it and how the stack pointer points to the top-of-stack.

Figure 4.2 shows that the top-of-stack has the lowest address. This may seem an upside-down approach, but some people may find this quite natural. However, it is one which is convenient for the operating system. As we add numbers to the stack, or take numbers from it, the computer alters a variable called the *stack pointer*. This variable is stored at another part of memory and 'points at' the top-of-stack. In other words, the value of the stack pointer is the address of the memory location which is currently the top-of-stack.

There is a word which lets us find out the value of the stack pointer. This word is called 'S–P–fetch' (see Fig. 4.3). In Fig. 4.3 we

```
SP@ .
88 OK

1 2 3 SP@ .
82 OK

4 5 SP@ .
78 OK


. . . . SP@ .
5 4 3 2 86 OK
```

Fig. 4.3.

began with an empty stack. The stack pointer equalled 88. The word 'S–P–fetch' puts the value of the stack pointer at top-of-stack. The value it puts is the value of the stack pointer *before* 'S–P–fetch' acted. If we follow 'S–P–fetch' with 'dot' to display this value, this removes the value from top-of-stack, leaving the stack as it was before.

The next step in Fig. 4.3 was to add three more numbers to the stack, then find the value of the stack pointer. This becomes 82. We put three numbers on the stack, but the pointer moved six locations further down in memory. It seems that it takes two locations (two bytes) to store each number. This is the stage illustrated in Fig. 4.2.

To check on this we put two more numbers (4 and 5) on the stack. The stack pointer changed to 78, showing that four more bytes were used. Finally, we removed four numbers from the stack by using 'dot'. These numbers were displayed and, after them, came the latest value of the stack pointer, 86. Removing four numbers from the stack shifted the pointer eight places along in memory.

It is clear from the above that the stack uses two locations in memory for storing each number. Each 'box' in Fig. 4.2 represents

two cells of the computer's memory. This means that it is possible to store a 16-digit number at each position on the stack. The largest possible 16-digit binary number is 1111111111111111, equivalent to 65535 in decimal. It is therefore possible to store numbers up to this size in each location of the stack.

Normally, numbers are not stored this way. Instead the computer uses only 15 of the 16 digits for storing the number itself. This allows numbers up to 32767, which is enough for most purposes. The 16th digit is used for giving the number a *sign*, to indicate whether it is positive or negative. If the 16th digit is 0, the number is positive; if it is 1, it is negative. The system works as in Table 4.1.

Table 4.1

Binary number	*Decimal equivalent*
0000000000000000	0
0000000000000001	1
0000000000000010	2
0000000000000011	3
0000000000000100	4
...	...
0111111111111100	32764
0111111111111101	32765
0111111111111110	32766
0111111111111111	32767
1000000000000000	−32768
1000000000000001	−32767
1000000000000010	−32766
1000000000000011	−32765
...	...
1111111111111100	−4
1111111111111101	−3
1111111111111110	−2
1111111111111111	−1
0000000000000000	0

It is easy to understand the top half of this table. The 16th digit (the one on the extreme left) is 0, so all the numbers are positive. They run from 0 up to the largest possible with 15 bits, 32767. Adding 1 to

this number gives the next number in the table, 1 followed by 15 zeros. Because this begins with 1 it is considered by the computer to represent a negative number. But why is it −32768, instead of −0?

This is better explained if we go to the end of the table and work backwards. The lowest line of the table has all zeros. Getting to this line from the line just above it is like clocking up the last mile on a car mileometer which registers up to 99999 miles. The next mile takes it to 00000 miles, as if it had never been driven since it was made! The sixth digit carried over simply does not exist for there is no room for it on the dial. Similarly, counting one more after 1111111111111111 gives all zeros. Conversely, counting one *back* from zero gives 16 ones. Sixteen 1s must be equivalent to −1 (one step backwards). As we read the table from the bottom line upward, the values are 0, −1, −2, −3 and so on until we eventually reach −32768. Going one more line up the table, we return to the region in which the 16th digit is a zero and we are back in positive numbers again.

As explained above, the computer normally treats the 16-bit numbers as *signed* numbers. This method of storing negative numbers makes use of what is called the *two's complement*. Here is an easy way to find how any negative number is stored. Take as an example, the number −56:

Write +56 in binary		0000000000111000
Write all 0s as 1s and all 1s as 0s		1111111111000111
Finally, add 1	+	1
		1111111111001000

This gives the way that −56 is stored.
Let us try this again for −32768:

+32768 in binary is		1000000000000000
Invert it		0111111111111111
Add 1	+	1
		1000000000000000

This confirms what we have seen in the table above.

It is usually more convenient to have the computer dealing with signed numbers, so that we have negative numbers when we want them. But sometimes it is better for the computer to treat the numbers as *unsigned*, with values ranging between 0 and 65535, and with no negative numbers allowed. If this is what is required, there are FORTH words to make the computer treat the numbers as unsigned.

To summarise

In this chapter you have found how to:

- Use the stack pointer to investigate the way the stack is used.
- Write a decimal number as a signed binary number.

You have used this FORTH word:

- SP@ 'S-P-fetch' (... address of stack pointer).

You have learned that:

- The stack is a special section of memory.
- Values are stored there as bytes, consisting of 8 binary digits, or bits.
- FORTH usually stores numbers as single-precision numbers each occupying 2 bytes on the stack.
- Single-precision numbers can have any value in the range −32768 to 32767.
- FORTH normally deals only with integer values.

Explore more

Try addition and subtraction of binary numbers. First make the computer display the binary equivalents of the decimal numbers given below. Write these down. Then add or subtract them, as indicated, using the following rules of binary arithmetic:

0 + 0 = 0	0 − 0 = 0
0 + 1 = 1	0 − 1 = 1, borrow 1
1 + 0 = 1	1 − 0 = 1
1 + 1 = 0, carry 1	1 − 1 = 0

Here are the decimal numbers to work with:

4 + 5	4 − 5
255 + 1	128 − 1
256 + 256	10 + 21
10 − 2	− 32768 + 1

When you have worked out the answers in binary on paper, use the computer to check your results.

Chapter Five
Numbers In Store

Suppose that you are planning your holiday in a foreign country. You have a travel guide which gives the costs of meals, hotel rooms, petrol and other essential costs in terms of the local currency. You want to see what these prices are when converted to pounds. The local currency is the franc, and your travel agent tells you that the exchange rate is 12 francs for 1 pound. Although exchange rates can vary, sometimes dramatically, you will want to use the same rate for all the calculations you do. For this reason, we regard the rate as a *constant*. FORTH has a special way of defining constants:

```
12 CONSTANT RATE
OK
```

The word CONSTANT has defined a constant called RATE and given it the value 12.

We used a *defining word* called 'colon' (:), in earlier chapters. A defining word is used to define other FORTH words. CONSTANT is another kind of defining word. Like 'colon', it defines other FORTH words but unlike ' colon' the words it defines are all the names of stored numbers. CONSTANT expects to find a value on the stack, ready to be assigned to the named constant. After a constant has been defined by using CONSTANT, we use the name of the constant whenever we want the value of the constant to be put on the stack. Try it:

```
RATE .
12 OK
```

Let's use CONSTANT to find out how many pounds a meal costs, if it costs 50 francs in local currency. The calculation required is to divide the cost in francs by the rate of exchange:

```
50 RATE / .
4 OK
```

This line introduces a FORTH word called 'divide'. This uses the same symbol as is used in BASIC. The action of the line above is to put the value 50 on the stack, followed by the value of RATE (12). Then 'divide' divides second-on-stack (50) by top-of-stack (12), giving the answer, 4, which is displayed by 'dot'. You may object that 50/12 gives 4.16666667. This is true, but remember that FORTH deals only in integers. It ignores the figures after the decimal point. You may think it most inconvenient of FORTH to drop the pence, but it is no more inconvenient than BASIC, which insists on giving you *all* the decimal places, as in the 4.16666667 quoted above. With either language we need a little more programming to get exactly what we want.

If you want to use FORTH in a currency conversion application, you will need to work out ways of making it handle the pence as well as the pounds. One way to do this is explained later in the chapter. For the moment, we will ignore the odd pence and look further at the uses of CONSTANT. Let us suppose that the travel guide told you that a hotel room costs 120 francs a night. This too is a constant, so we can define:

```
120 CONSTANT ROOM
OK
```

The two constants may now be used to calculate the cost of staying in the hotel for different numbers of nights. If your stay is for 7 nights:

```
7 ROOM * RATE / .
70 OK
```

Let us look at the stages of this calculation. First we put 7 on the stack, followed by ROOM (120). The next word is 'times' (*). As might be expected, the action of this word is to take the top two numbers from the stack, multiply them together, and place their product (the result) on the stack. At this stage, top-of-stack is 840. Then we put RATE (12) on top-of-stack. The next word, 'divide', takes the top two numbers off the stack, divides them (840/12) and leaves the answer (70) on the top-of-stack. This is displayed by 'dot'.

If you are not clear about what happened, Fig. 5.1 illustrates the same sequence, but with 'dot-S' included after each word, to make the computer print out the contents of the stack at every stage. Whenever you have difficulty in following how a line of FORTH works, make use of 'dot-S' as in Fig. 5.1. We shall not use it again for this purpose in this book, but will assume that *you* will

```
7 .S ROOM .S * .S RATE .S / .S .
7
7 120
840
840 12
70 70 OK
```

Fig. 5.1.

use it whenever you want things made clear. If your version of FORTH does not include 'dot–S' or its equivalent, you may be able to use the word as defined on page 167.

In the example above we used the two words, 'times' and 'divide'. Compare the line above with the line below:

```
7 ROOM RATE */ .
70 OK
```

Instead of using the words 'times' and 'divide' (with the required space between them, of course) we have used a new word, called 'times-divide'. This has the same action as the two separate words, but has an advantage. The two numbers that are multiplied together are stored as a *double-precision number*, which is then divided by the third number. A double-precision number is stored in *four* bytes (32 bits) instead of two, which means that very large numbers can be stored precisely. Such numbers can range between −2147483648 and +2147483647. If we use the first method, with separate times and divide, the result of multiplication is stored only as a single-precision number. It might easily happen that multiplying two large numbers gives a result bigger than the largest possible single-precision number. This would lead to a wrong result. Using 'times-divide' with its double-precision intermediate stage means that we always get a precise result, unless the numbers used are very great indeed. More about double-precision calculations in Chapter Eleven.

As another example, here is a word to convert temperatures in degrees Celsius to temperatures in degrees Fahrenheit:

```
: FAHR 9 5 */ 32 + . ;
```

We put the Celsius temperature on the stack, then use FAHR. It multiplies the temperature by 9, divides it by 5 and adds 32. This gives the temperature in Fahrenheit:

```
21 FAHR
69 OK
```

Look after the pence ...

Now to return to our currency conversions. It is all very well for the computer to ignore the pence and tell us only about the pounds, but pence are important. If the average cost of a meal is 57 francs the cost in pounds is £4.75. But the computer works out the cost as £4. Every meal costs 75p more than the computer tells us, and we might have no money left for meals during the last few days of the holiday! There is a word in the line below that helps us look after the pence:

```
57 RATE /MOD . .
4 9 OK
```

The action of 'divide-mod' is the same as that of 'divide' in that it performs a division but, before it puts the result (the *dividend*) on the stack, it puts the *remainder* there. The example above shows that 57 divided by 12 (=RATE) gives 4, remainder 9. The 9 indicates that there are 9 francs remaining, which must now be converted into pence. The way to do this is to multiply the number of francs remaining by 100 and divide by RATE. This is another use for 'times-divide'. The example above shows that the number of pounds was on top-of-stack. We display this so as to get it out of the way while the pence are calculated.

```
57 RATE /MOD . 100 RATE */ .
4 75 OK
```

So far, so good! We have the figures we need, but it would be better if they could be displayed in the more usual format, as '£4–75'. It is better to keep both values on the stack, then use a routine to display them with the '£' and '–' in the right places.

In the line above, we displayed the value for pounds, so as to get it off the stack and out of the way while we worked out the pence. There is another way of getting the pounds value out of the way without taking it off the stack. This uses the word SWAP. As its name implies, SWAP swaps the top two values on the stack. After '57 RATE /MOD', the pounds are at top-of-stack and the remainder is at second-on-stack. SWAP tucks away the pounds safely at second-on-stack, and brings the remainder to top-of-stack, ready to be dealt with. We define a new word which uses SWAP to do this:

```
: POUNDS RATE /MOD SWAP ;
```

This leaves the remainder at top-of-stack. Here is a word to convert the remainder to pence:

```
: PENCE 100 RATE */ ;
```

One of the good things about FORTH is that you can try out your words as you define them, to see if they really do what you want them to do. Try out new words, POUNDS and PENCE:

```
57 POUNDS PENCE SWAP . .
4 75 OK
```

This gives the same result as the earlier line. Note that we have used SWAP again. After POUNDS and PENCE, the pounds were at second-on-stack (where POUNDS put them) and pence were at top-of-stack, as left by PENCE. We need to SWAP again to bring the pounds back to top-of-stack ready for displaying.

The values are in the right order for displaying. Now we have to add the '£' and '–'. There is a word for displaying strings of characters, called 'dot-quote' (."). This works rather like PRINT in BASIC. It does not make use of the stack, but displays a string on the screen immediately. The end of the string is marked by a second quote character. Here is 'dot-quote' in action:

```
57 POUNDS PENCE SWAP ." £" . ." -" .
£4 -75 OK
```

The values are ready on the stack after SWAP. Then 'dot-quote' displays a '£'. Note the *space* after 'dot-quote'. All FORTH words must have a space after them. This space is not displayed. The string contains only the single '£' symbol. The quote which ends the string comes immediately after the '£' (unless you would prefer a space to be displayed after the '£'). After using 'dot-quote' to display '£' we display the pounds, using 'dot'. Then we use 'dot-quote' again to display the '–', and finally use 'dot' to display the pence.

All is well, but not quite. When FORTH displays a number it puts a blank space after it. To get rid of the space after the 4 we use another word, 'dot-R'. This sets up a *field* (think of it as a row of blank spaces) and prints the number at the right-hand end of the field. In other words, the number has no space after it. Before using 'dot-R' we must put a number on the stack to tell it how many spaces the field is to have. Since we have only one digit in the pounds, we want only one space. This displays the pounds with no space before it or after it. If the number of pounds happens to have two or more digits, 'dot-R' makes the string just long enough to hold the extra digits.

The line of the previous example was getting rather long, so it is time we defined a new word. Here is the word which prints out the result in the format we require:

```
: FORMAT ." £" 1 .R ." -" . CR ;
```

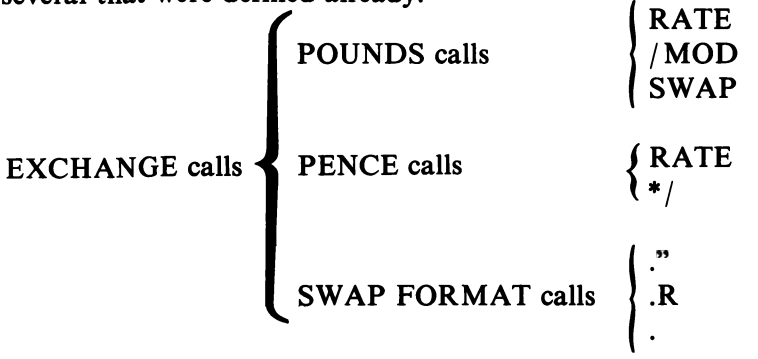
You can see that this is similar to the last section of the previous line, except that we have put '1 .R' in place of the first 'dot'. The 1 goes on the stack to tell the computer that 'dot-R' must make its field only one character wide. This 1 is removed from the stack when 'dot-R' acts. There is one further improvement. The definition ends with the word 'CR' which is short for carriage return. It is the equivalent of pressing the RETURN key at the end of the line. The effect of this is to make the computer go on to the next line of the screen before it prints its customary 'OK'. This makes it easier to see the result of the calculation. Now we are ready to combine all our previously defined words into a single word, EXCHANGE:

```
: EXCHANGE POUNDS PENCE SWAP FORMAT ;
```

This performs the whole operation of converting francs to pounds and pence and displaying the results in the required format:

```
57 EXCHANGE
£4-75
OK
```

By keying in the number of francs and using the single word EXCHANGE we are able to call upon all the words we have defined and several that were defined already.



Some of the words originally there, such as SWAP, may call other words, and these in turn may call upon machine code routines to perform the actions we required. All of this can be put into operation by the single word EXCHANGE. This example illustrates the threaded nature of FORTH.

Before going further, try using EXCHANGE with various values and check that it always performs the calculation as it should.

If you like, you can put it into a loop, to calculate the equivalents for whole numbers of francs over a given range:

```
: TABLE 20 1 DO I DUP . EXCHANGE LOOP ;
```

There are some words in FORTH which cannot be executed immediately. They can be used only in a word definition. DO and LOOP are such words. The word above sets the end of the loop at 20 and the beginning at 1. The action that is to be repeated is placed between DO and LOOP. The first action there is the word I. This places on the stack the loop number (or index) each time through the loop. Thus I puts 1 on the stack the first time round, 2 on the second time round, and so on. We need to have I for two reasons. One is that we want to display it each time round, so that the table shows the number of francs corresponding to each value in pounds. The second reason is that we require I so that EXCHANGE has something to operate on. If we put I on the stack, and then display it, using 'dot', we shall of course lose I from the stack. We need *two* Is. These we obtain by using DUP, which is short for duplicate. Using I puts I at top-of-stack. Following this with DUP puts another I on

```
TABLE
1  £0-8
2  £0-16
3  £0-25
4  £0-33
5  £0-41
6  £0-50
7  £0-58
8  £0-66
9  £0-75
10  £0-83
11  £0-91
12  £1-0
13  £1-8
14  £1-16
15  £1-25
16  £1-33
17  £1-41
18  £1-50
19  £1-58
OK
```

Fig. 5.2.

top-of-stack, with the original I at second-on-stack. 'Dot' displays I from top-of-stack, leaving the other I (now at top-of-stack) for EXCHANGE to work on. All of this may now be called into operation by simply typing TABLE (see Fig. 5.2).

You get a table of conversions of francs to pounds for every value of francs from 1 to ...? Why not 20? This is a feature in which loops in FORTH differ from loops in BASIC. They stop when the index (I) reaches the upper limit (20 in this case) and are not executed for the value of the upper limit. If you want a table from 1 to 20, you must set the upper limit to 21. In general, set it to 1 more than you need.

PENCE, and all the words which use it, does not give a completely accurate answer. Like our original version of POUNDS, it ignores fractions. This is not really of any consequence, but if you would prefer to have the amount rounded off to the nearest whole penny, see Chapter Eight.

Variables

Although the rate of exchange may be a constant, in that you want to use the same figure in a series of calculations, experience tells us that rates of exchange vary from day to day. In this respect it is a *variable*. FORTH has another defining word. VARIABLE, for storing variable numbers:

```
VARIABLE RATE
```

Unless you have switched off the computer or restarted FORTH since the last session, you will see 'MSG #4' telling you that you are using the same name as you used for the constant. Take no notice of this.

Note that in this definition, we do not have to give a value to RATE. To find out what value it already has, try this:

```
RATE .
15759 OK
```

This is an odd value to have! What we see here is not the value of RATE, but the address in memory at which the *value* of RATE is stored. This is one way in which VARIABLE differs from CONSTANT. Whereas CONSTANT puts the value of the constant on the top-of-stack, VARIABLE puts the *address* at which the value is stored.

There is a word which will let us find out this value. Appropriately, it is called 'fetch'. Its symbol is '@' and it fetches the value out of store:

```
RATE @ .
0 OK
```

The action of 'fetch' is to take an address from the top-of-stack and place on top-of-stack the value stored at that address. In the line above, 'fetch' finds '15759' at top-of-stack, looks at that address, finds zero stored there and places zero at top-of-stack. This shows us that a newly defined variable has value zero. To assign a value to RATE or any other variable, we use a word which has the opposite action to 'fetch'. This word is called 'store' and has the symbol '!':

```
12 RATE !
OK
```

We first place on the stack the value which is to be given to the variable. Then the word RATE places the address of RATE on the stack. 'Store' takes the address from top-of-stack, and the value from second-on-stack, and stores the value at the address. Use the line given earlier to check that the value of RATE has now changed to 12.

If we want to be able to use a varying exchange rate in our calculations, it is more convenient to use VARIABLE than CONSTANT. It *is* possible to change the value of a constant, but this is less easy to do. In any case, the point about a constant is that it does not change!

When POUNDS and PENCE were defined above, RATE was a constant. If you use POUNDS and PENCE now, they will still use the old definition of RATE as a constant. To get these words to make use of your new definition, you will have to redefine them:

```
: POUNDS RATE @ /MOD SWAP ;

: PENCE 100 RATE @ */ ;
```

These are the same as the previous definitions, except that we have inserted 'fetch' because RATE is now a variable. A quick check shows that they are working properly:

```
57 POUNDS PENCE SWAP . .
4 75 OK
```

We also need to redefine EXCHANGE, so that it uses the new definitions of POUNDS and PENCE:

```
: EXCHANGE POUNDS PENCE SWAP FORMAT ;
```

It is not necessary to redefine FORMAT since this does not rely on anything that has been changed.

The daily exchange rate may be used in this application:

```
: DAILY RATE ! EXCHANGE ;
```

This expects to find the rate at top-of-stack and the amount to be converted at second-on-stack. It first puts the address of RATE on the stack, then stores the value of the rate at the address of RATE. Then EXCHANGE works out the amount in francs, using the amount which it now finds at top-of-stack. If you find this difficult to follow, redefine DAILY with 'dot-S' inserted after each word. Figure 5.3 shows it in action.

```
57 13 DAILY
£4-38
OK

57 11 DAILY
£5-18
OK
```

Fig. 5.3.

In this chapter we have seen how CONSTANT is used for storing numbers which are to remain unchanged during the running of an application, while VARIABLE is used for storing numbers that are to change. In the course of this discussion we have looked at many other useful words in FORTH. If you are going straight on to the next chapter, with perhaps only a short break for a cup of tea, leave the computer switched on.

To summarise

In this chapter you have found out how to:

- Define and use constants and variables.
- Make the computer display messages on the screen.
- Format the display of numbers.

You have used these FORTH words:

- / 'divide' (n1\n2 ... n1/n2).

- * 'times' (n1\n2 ... n1*n2).
- */ 'times-divide' (n1\n2\n3 ... (n1*n2)/n3).
- /MOD 'divide-mod' (n1\n2 ... remainder\quotient of n1/n2). The remainder has the same sign as n1.
- SWAP (n1\n2 ... n2\n1).
- @ 'fetch' leaves value n found at address (address ... n).
- ! 'store' stores value n at address (n\address ...).
- ." 'dot-quote' displays text following ." as far as the next quote mark (...).
- .R 'dot-R' displays number at right-hand side of a field which is n spaces wide (n ...).
- CR causes a carriage-return (...).
- I puts the index of DO...NEXT on the stack (...I).
- DUP (n ... n\n).
- CONSTANT defines constant of value n (n ...). When the constant name is used its stack action is (... n).
- VARIABLE defines variable with initial value zero. When the variable name is used, its stack action is (... address at which the value is stored).

You have learned that:

- FORTH has defining words, such as CONSTANT and VARIABLE, used to define other FORTH words.
- A double-precision number is stored in 4 bytes.
- A double-precision number can have any value in the range −2147483648 to 2147483647.

Explore more

(1) Define a word, MM, to convert lengths in inches to lengths in millimetres.

(2) Define a word, MPS, to convert speeds in miles per hour to speeds in metres per second.

(3) The cost of a ball-point pen is 15p, excluding VAT. Define a word, using DO...LOOP, to display a table to show the cost of any number of pens from 1 to 10, in pence, excluding and including 15% VAT.

Chapter Six
See How They Run

This chapter is an exploration of animated graphics. From the beginning, it is clear that we have several problems:

(1) No two makes of micro handle their graphics in exactly the same way.

(2) No two makes of micro have exactly the same version of FORTH.

(3) The FORTH standard does not include special graphics commands.

The third point is to be expected, for a standard system cannot provide for non-standard features of each brand of micro.

Taken together, these difficulties might seem to be insurmountable. But the flexibility of FORTH comes to our aid, making graphics designing easy on almost any micro. We shall need to plan how to make best use of FORTH for this purpose. This involves taking rather the opposite approach from that of Chapter Five. In that chapter we wandered through a series of words, gradually improving upon each until we obtained the exchange application that we wanted. In this chapter we must be more systematic, which is what one should be when writing in FORTH.

First we will set out our requirements. We must be able to:

(1) Define graphics characters to our own design.
(2) Make any character appear anywhere on the screen.
(3) Replace any character on the screen by a different one, or move a character from one part of the screen to another.

To avoid undue complications, the method used for specifying any given point on the screen is to state its column and row numbers. The column number is represented by X. Many computers have a 40-column screen, so X ranges from 1 to 40, or from 0 to 39,

depending on the micro. If your computer has a different number of columns, there is no difficulty in allowing for this. The row number is represented by Y. This often ranges from 1 to 24, or 0 to 23. In some computers, Y increases going *down* the screen, in others it increases going *up*. This is something else that is easy to allow for later. In short, screen location is specified by X and Y, just as it might be in the BASIC keyword, TAB(X,Y).

Now to define a few words to act as an 'interface' between standard FORTH and the particular graphics system of your micro, then your micro will be able to operate with the graphics words described later.

Looking slightly ahead, it is clear that eventually we shall want to be able to have several, perhaps dozens of characters on the screen, all being moved around independently. We shall want to know where each one is at any given time. We need a list of the characters, in order, giving the X and Y positions of each. The list is to be held in the computer's RAM. We need to be able to refer to any entry by number, and to alter any entry as often as we wish. Readers who are familiar with BASIC will recognise that what we want is an *array*. FORTH does not provide arrays but, as usual, we can readily add them to the system.

When we defined a constant or a variable in Chapter Five, we had ready-made defining words, CONSTANT and VARIABLE. There is no defining word ARRAY in FORTH, so we will have to define one ourselves. Note that we are not simply defining *a word*, as we have done before, by using 'colon' ,CONSTANT or VARIABLE. We are defining *a defining word*. In 79-FORTH, we use a pair of words for this purpose, CREATE and 'does-greater'. The latter is keyed in as DOES>. Here is the definition of ARRAY (see Appendix A for the Jupiter Ace version):

```
: ARRAY CREATE 2 * ALLOT
        DOES> SWAP 2 * + ;
```

The definition begins with 'colon' just like an ordinary word definition. Then comes ARRAY, the name of the word we are going to define. There are two aspects to a defining word:

(1) How is it to act when it is doing its defining?
(2) How is the word it defines to act, when it is used in an application?

The definition above is in two parts. The part beginning with CREATE states how the defining word is to act: it says how an array

is to be created. The part from DOES> to the end of the definition says how the arrays are to act when called.

An array needs a name, so that we can call it up. It also needs some memory so that it can store data. CREATE will set aside a block of memory. The name we shall give to the array will be stored at the beginning of this block. The sequence 2 * ALLOT, expects a value on the stack to say how many numbers are to be stored in the array. This number is multiplied by 2 to give the number of bytes required, then ALLOT sets aside that number of bytes for the array.

Before going any further here is how we use ARRAY:

```
8 ARRAY CHARS
```

The figure 8 indicates how many numbers are to be stored. CHARS is the name we are giving to the array. It is short for 'characters', since it will be used for holding details of graphics characters. When it is defined, the values in CHARS are the values already present on the block of memory that happened to be used. Try reading one of the values (see Fig. 6.1). The locations in the array are numbered

```
3 CHARS @ .
14 OK
```

Fig. 6.1.

from 0 to 7, so this tells us what is in the fourth location (Fig. 6.2). Let us see exactly what happened in the line above. We put the number of the location on the stack. CHARS puts the address of the first memory location of the array on the stack (just as using VARIABLE puts on the stack the address at which the single value is stored).

Now the DOES> part of the definition comes into action. Here is the state of the stack (top-of-stack on right), stage by stage. We are using the example above and assume the address of the start of the memory block is 15504

We start with	3	15504	
SWAP	15504	3	
2	15504	3	2
*	15504	6	
+	15510		

This DOES> part of the definition of ARRAY calculates the address of any given location in the array and leaves this address on the stack. Then we use 'fetch' to take that address from the stack and

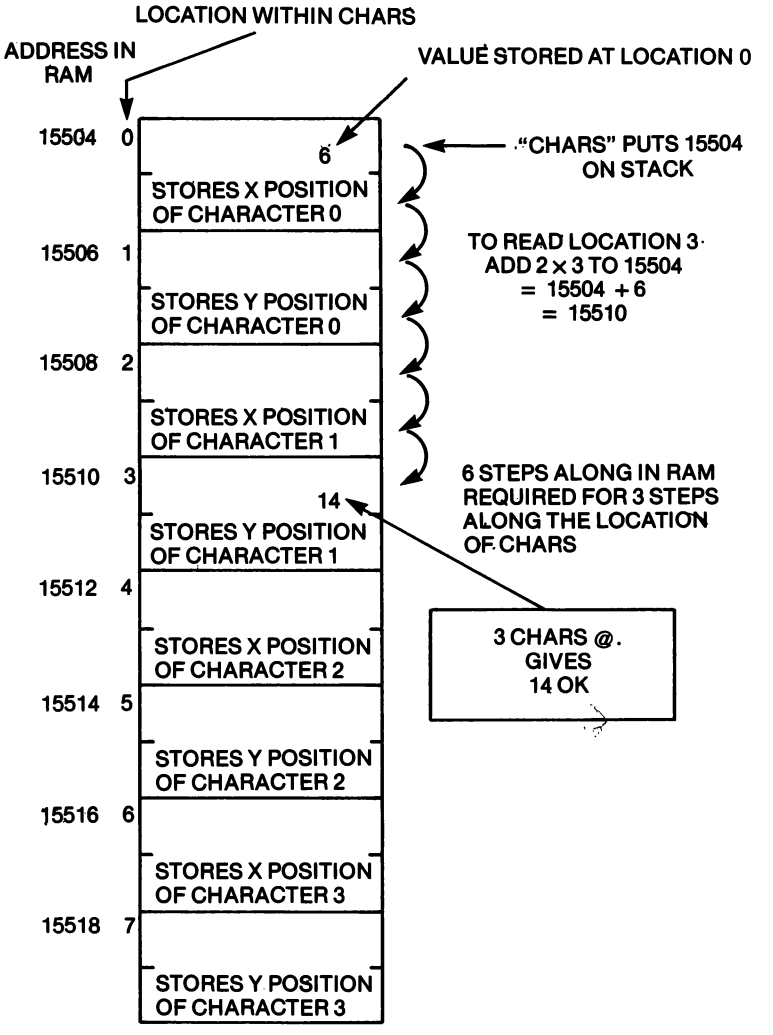


Fig. 6.2. The region of RAM which is allotted to the array CHARS. Although each location in CHARS requires *two* bytes of RAM, each location holds only *one* value.

replace it with the value stored at that address.

If you would like to see the address left by CHARS *before* the values are fetched, use:

```
3 CHARS .
15504 OK
```

Try the above for several locations, you will see that successive locations begin 2 bytes further along in memory. This is because

FORTH needs two bytes to store each number, just as it does when storing numbers on the stack. Storing a value in an array is done like this:

```
23 4 CHARS !
OK
```

The first value is the amount to be stored, the second is the location in which it is to be stored. CHARS leaves the address of the location on top-of-stack, as above, and the value to be stored is second-on-stack. 'Store' then uses these two numbers to store the value in the array.

After that flight into the higher realms of FORTH, we return to the subject of graphics displays.

User-defined graphics

Most micros allow the user to define their own graphics characters. The way this is done depends on the micro itself, though the principle is generally the same for all. To define a character such as a mouse, for example, we begin by sketching the design on an 8-by-8 grid (Fig. 6.3). Each row of the grid is coded, by considering the shaded

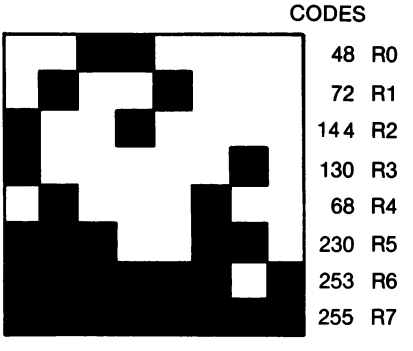


Fig. 6.3. Design for a mouse character.

squares to represent '1' and the unshaded squares to represent '0'. The 8 digits of each row are taken to be an 8-bit number, in binary. Some micros allow you to enter the binary numbers directly (actually FORTH allows this, as explained in Chapter Eleven), but we shall use decimal numbers here. We will refer to these numbers as the row codes, R0 to R7.

The next few paragraphs may send you off to the manual which came with your computer, or the manual belonging to your version of FORTH to find out exactly what to do on your computer. The two examples below are intended to show you the way to set about the task. First of all, we must specify exactly what is to be done. The aim is to define a word, CHARDEF, which defines a graphics character and which has the following stack action:

(R7\R6\R5\R4\R3\R2\R1\R0\N ...)

The row of symbols above means that CHARDEF expects to find the row codes, in decimal, in *bottom-to-top* order, with the code number of the character (N) on top-of-stack. We shall be using several characters, and these are to be numbered from 0 upward. The absence of anything after the '...' indicates that CHARDEF is to leave nothing on the stack when it has finished.

Here is a word in Acornsoft FORTH which acts as described above:

```
: CHARDEF 23 >VDU 224 + 9 0 DO >VDU LOOP ;
```

It uses the VDU commands of the BBC Microcomputer and Electron. The word >VDU is special to this version of FORTH, so is not likely to work with any other version. However, most FORTHs have words to do similar things, that are adapted to the computer. This line begins by putting 23 on the stack, then sending this value as a VDU 23 statement, which tells the computer that a graphics character is being defined. Then it sends the character code number. On the BBC Microcomputer, characters are coded from 224 to 255. The line adds 224 to the code number which is at the top-of-stack. This means that we can use character codes numbered from 0 to 31. The definition ends with a DO ... LOOP which takes the altered character code number and then the row codes from the stack and sends them to the VDU 23 routine. Note that the limits of the loop are 0 to 9, giving 9 repetitions of the loop. Here is CHARDEF in action defining character 0 as a mouse:

```
255 253 230 68 130 144 72 48 0 CHARDEF
```

A version of CHARDEF for the Jupiter Ace is given in Appendix A.

Displaying a character

Once again, most versions of FORTH will have some special way of printing a character at any given location on the screen. Below we use the Acornsoft words to define a word PLACEIT. You will need to adapt any words given in your FORTH to perform the following stack action:

(N\,X\,Y...)

PLACEIT expects to find the character code number and the required X (column) and Y (row) positions. In the Acornsoft version we use the VDU 31 statement:

```
: PLACEIT 31 >VDU SWAP >VDU >VDU
          224 + >VDU ;
```

This first puts 31 on the stack then sends this as VDU 31. X and Y are SWAPped, so that it sends X, then Y. This leaves N on top-of-stack. This has 224 added to it to obtain the correct character code which is also sent as part of the VDU statement. This VDU 31 statement is the equivalent of

'TAB(X,Y);CHR$(N)'

Note that this does not work when the computer is in Mode 7. Mode 4 is a good one for use during this chapter. PLACEIT can be defined rather more simply on the Jupiter Ace, as shown in Appendix A.

PLACEIT is to be used for putting a graphics character anywhere on the screen. See it in action:

```
0 20 15 PLACEIT
```

Character 0 (the mouse) appears near to the middle of the screen. Try this with other values for X and Y. You could also define some other characters (numbered 1, 2, up to 7) and put these in various parts of the screen. Do not worry about the 'OK' which always appears to the right of the mouse. We will eliminate this later.

Ready for off

You should now have a means of defining arrays on your computer, and of defining graphics characters. You should also have a word for placing any character at any part of your screen. Your words may have been defined differently from those above but they do the same

things. You are now ready to go ahead making up other words. But, from now on, whatever FORTH or micro you have, the applications in the rest of this chapter should work. The only adjustments you might need to make are to allow for the number of rows and columns on your screen.

If we are intending to place a lot of characters on the screen at one time, we want to know where each one is. Here is a word to do this:

```
: STORPLACE DUP 4 PICK 2 * 1+
            CHARS ! OVER 4 PICK 2 *
            CHARS ! PLACEIT ;
```

We call it STORPLACE because it first picks up the X and Y positions from the stack and stores them in the array CHARS, then it uses PLACEIT to place the character on the screen. Like PLACEIT, the stack action of this word is:

(N\X\Y...)

You will notice two new words in this definition, 'one-plus' and PICK. 'One-plus' increments top-of-stack by 1; it has the same action as the two words '1 +'. The action of PICK is illustrated in Fig. 6.4 along with the action of several other words which shift values around on the stack. In STORPLACE we use '4 PICK' to find the value which is fourth-on-stack and place it at top-of-stack. STORPLACE has a rather long definition; this is what happens, stage-by-stage. As before, we show the stack with top-of-stack to the right:

We begin with	N X Y
DUP	N X Y Y
4 PICK	N X Y Y N
2 *	N X Y Y 2N
1+	N X Y Y 2N+1
CHARS !	N X Y
OVER	N X Y X
4 PICK	N X Y X N
2*	N X Y X 2N
CHARS !	N X Y

The stack is unaltered and is ready for the final word, PLACEIT. Note that before using CHARS we have to double the value of N, since there are two values (X and Y) associated with each character (Fig. 6.2). Before the first CHAR we have to add one to twice N so

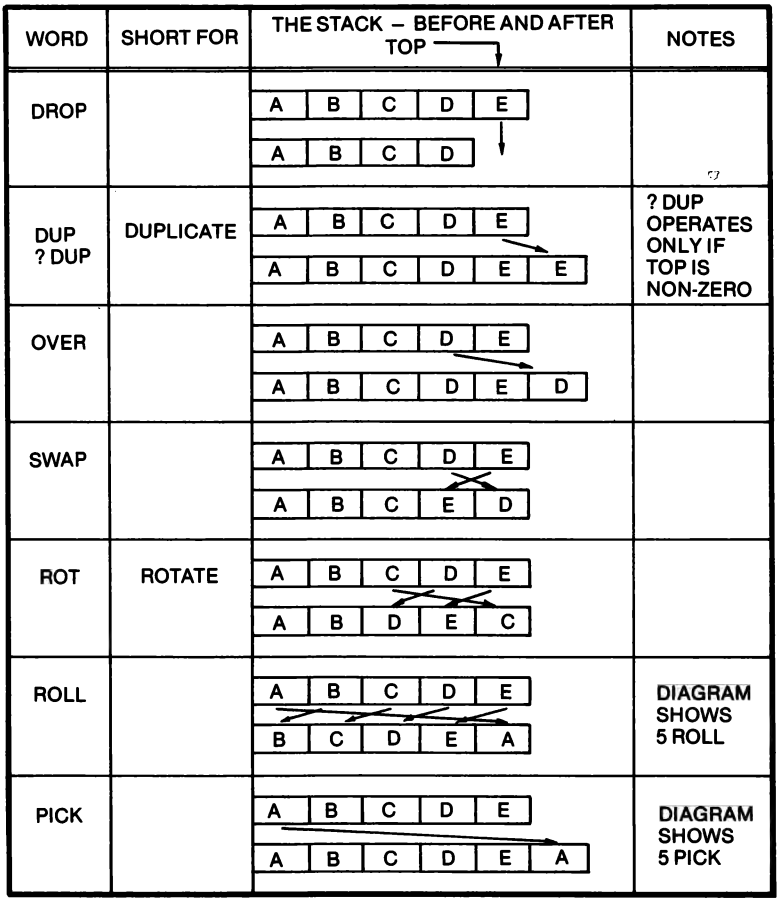


Fig. 6.4. Stack operators. The letters A — E represent numerical values already on the stack.

that the value of Y is stored as the second of these two numbers. A little confusion may arise here. We have already said that an array uses two *bytes* to store each number. Now we are saying that two *numbers* have to be stored (X, Y) to locate each character.

Now is the time to have some more fun making the mouse appear in any part of the screen:

```
0 8 18 STORPLACE
```

Looking at the stack operations set out above, some readers may despair of ever working out word definitions for themselves. Here are a few tips on how to set about this task.

Writing definitions

Writing a definition is rather like those word puzzles in which you are given a starting word and a finishing word and asked to fill in all the words between, changing only one letter at a time:

FORTH
. . . . .
. . . . .
. . . . .
. . . . .
SITES

The intervening words are FORTS, SORTS, SORES, and SIRES. If you are the sort of person who loves solving these puzzles, FORTH holds no terrors for you! Begin by writing out the initial state of the stack, and also what you want it to be when a critical operation is to be performed. In STORPLACE, the first operation is to store Y in CHARS. We begin with (N\X\Y...) and need to get to (N\X\Y\2N+1 ...). Here it is, set out as a puzzle:

N X Y

N X Y Y 2N+1

We do not know how many steps the puzzle is going to take. When deciding what must be on the bottom line we remembered that CHARS removes both Y and 2N+1 from the stack, so we must retain the value of N, ready for storing X. The best course seems to keep it in its original place. We also need to have a copy of Y on the stack, ready for use by PLACEIT later on. These points decide what must be on the stack *just before* CHARS. Having established the beginning and end, we try working downward from the beginning and upward from the end – hoping to meet in the middle!

There is a clear way back from the end:

N X Y

2 * 1 + N X Y Y N
N X Y Y 2N+1

Where is the N to come from? We need to keep the original N on the stack so we must use a word such as PICK, DUP or OVER (see Fig. 6.4), which copies without removing the original number. The obvious choice is PICK:

```
                N X Y
4 PICK          N X Y Y N
2 : 1+          N X Y Y 2N+1
```

Now the rest is easy. To link the first line to the remainder all we need is DUP:

```
                N X Y
DUP             N X Y Y
4 PICK          N X Y Y N
2 * 1+          N X Y Y 2N+1
```

The link between beginning and end has been established. A similar line of reasoning can then be used to prepare the stack for the second occurrence of CHARS.

Moving graphics

Making a character appear to move across the screen is done by printing it in a given place, then blanking it out and reprinting it in an adjacent space. The character appears to move from the first place to the second place. If we repeat this action many times, the character can be moved across the entire screen. We already have a word, STORPLACE which stores the position of the character CHARS and then prints the character in that position. What we need next is a word to find out where the character is and replace it with a blank space. The word is called BLANKIT:

```
: BLANKIT -192 SWAP DUP 2 *
          CHARS @ SWAP 2 * 1+
          CHARS @ PLACEIT ;
```

This requires only N on the stack. It then finds the X and Y positions of character N from CHARS. The figure −192 requires some explanation. In the version of PLACEIT written for the BBC Microcomputer, 224 is added to the character code. The code actually sent to the VDU 23 statement is 224 greater than the code (N) that we use for numbering each character. The code for a space (actually its ASCII code) is 32, so to make PLACEIT display a space

we must supply it with a number which is 224 less than 32, which gives −192. Later, PLACEIT adds 224 to this, obtaining 32, and so printing a space. As an exercise in working out stack operations try setting out on paper what happens during the execution of BLANKIT.

If you are using a version of PLACEIT which adds some other amount to the character code (or perhaps uses it unchanged) substitute a suitable value for −192 in BLANKIT.

Try using STORPLACE, to display a character, then use BLANKIT to remove it:

```
0 12 6 STORPLACE BLANKIT
```

Finally, we need a word to find out where the character was *before* it was blanked, and then print it a specified number of places away. The word we use is MOVEIT:

```
: MOVEIT 3 PICK 2 * 1+ CHARS @
         + SWAP 3 PICK 2 * CHARS @
         SWAP STORPLACE ;
```

This requires the stack to hold:

(N\ difX\ difY ...)

DifX and difY are the amounts by which X and Y are to be changed. Here is another chance for you to see how SWAP and PICK are used. The first part of MOVEIT calculates the new positions by adding difX to X (got from CHARS) and difY to Y (also from CHARS). Then STORPLACE stores these new values in CHARS and prints the character in its new position. Now for a preliminary run:

```
: RUN 0 1 15 STORPLACE 39 1
      DO 0 BLANKIT 0 1 0 MOVEIT
      LOOP ;
```

The values in this word are set to make the mouse run from left to right across the 15th row of the screen. Whether it's the 15th up or the 15th down depends on how the display of your computer operates. The initial '0 1 15' places character 0 (the mouse) at column 1, row 15. You can alter these numbers to use a different character or to start it in a different place. The '39 1' are the end and beginning of the loop to move the mouse across the screen. If you have a different screen width, or want the mouse to run to a different position, you can alter the 39. Within the loop, BLANKIT blanks out the mouse ready for it to be displayed in its new position by MOVEIT. The

sequence '0 1 0' tells MOVEIT to move character 0 to the next column to the right (difX=1) but to stay on the same row (difY=0). The word leaves the mouse displayed at the right-hand side of the screen. Perhaps this is just as well, for FORTH works extremely quickly, making the mouse scamper across the screen so fast that you can hardly see it run!

Those who are used to BASIC programs may find it goes against the grain to make a program run more slowly, but it is obvious that we shall need to slow things down a bit here. The easiest way is to nest a DO ... LOOP inside the other one (see Fig. 6.5). RUNSLOW

```
: RUNSLOW 0 1 15 STORPLACE 39 1
        DO 0 BLANKIT 0 1 0 MOVEIT 1000 0
             DO
             LOOP
        LOOP 0 BLANKIT ;
OK
```

Fig. 6.5.

finishes with BLANKIT, to remove the last image of the mouse from the screen. There are endless variations on RUNSLOW. Try a few of them. Start the mouse from different parts of the screen and let it finish at different parts. Let it run up or down the screen, instead of across. Or let it take a diagonal path, instead.

Pause for breath

You now have all the essential words needed for producing moving graphics. With these, the screen can be filled with hordes of aliens from Outer Space, while you fire laser beams across the screen at them from a spacecraft. The fact that it was necessary to use a delay to slow the mouse down indicates that FORTH is capable of dealing with a screenful of objects at the speed required for action games. This is not the time to start designing such games, for there is more to find out about how FORTH takes decisions (for example, has the laser hit the alien?). This is explained in Chapter Eight. In the meantime, it is worth while saving the essential words on a FORTH screen, as has been done in Fig. 6.6.

```
SCR # 16       10 H
  0 ( GRAPHICS WORDS )
  1 : ARRAY CREATE 2 * ALLOT DOES> SWAP 2 * + ;
  2 8 ARRAY CHARS
  3 : CHARDEF 23 >VDU 224 + 9 0 DO >VDU LOOP ;
  4 : PLACEIT 31 >VDU SWAP >VDU >VDU
  5 224 + >VDU ;
  6 : STORPLACE DUP 4 PICK 2 * 1+
  7 CHARS ! OVER 4 PICK 2 *
  8 CHARS ! PLACEIT ;
  9 : BLANKIT -192 SWAP DUP 2 * CHARS @
 10 SWAP 2 * 1+ CHARS @ PLACEIT ;
 11 : MOVEIT 3 PICK 2 * 1+ CHARS @ +
 12 SWAP 3 PICK 2 * CHARS @  + SWAP
 13 STORPLACE ;
 14
 15
```

Fig. 6.6.

More mice?

When you are tired of mouse-running, why not invite all three blind mice on to the screen? This example shows a way of moving several characters at once. The application is designed to show the three blind mice chasing after the farmer's wife. First of all, define three graphics characters (0 to 2) as mice. Use the same design (Fig. 6.3) for them all, or work out a special design for each. Figure 6.7 shows the farmer's wife, who is defined as character 3.

Fig. 6.7 Design for the Farmer's Wife.

The wife is to move across the screen from left to right, followed by the three mice, one behind the other. There must be gaps between the mice. If a mouse is moved into the place just vacated by the

mouse in front of it, the illusion of motion is spoilt. To start the display, use STORPLACE in a loop:

```
: READY 4 0 DO I DUP 3 * 15 STORPLACE
            LOOP ;
```

The loop operates from 0 to 3. The loop index (I) is placed on the stack. Incidentally, if you have Acornsoft FORTH and have just used the Editor, the meaning of I will have been redefined. To recover the normal function of I, type FORTH. DUP makes a second copy of I, which is then multiplied by 3. Then 15 is put on the stack:

```
I  3*I  15
```

These values provide the N, X and Y required by STORPLACE. I ranges from 0 to 3, so that each character is displayed in turn. X (=3*I) ranges from 0 to 9, placing the mice and the wife at every third space. They are all on row 15. The next word makes all 4 characters move forward one step:

```
: STEP 4 0 DO I BLANKIT I 1 0 MOVEIT
          LOOP ;
```

This is a loop to first blank each character, then to re-display it one space to the right. As in READY, the loop counter I is used to determine which character is to be operated on.

So now we are ready to 'See How They Run':

```
: SHTR READY 30 0 DO STEP
                  LOOP 4 0 DO I BLANKIT
                           LOOP ;
```

STEP is enclosed in a loop which repeats 30 times. This takes the wife across the screen, followed by the trio of mice. The second part of SHTR is a loop to blank out the characters at the end of the run. They move quickly, and you could possibly insert a delay loop in SHTR. Or you could make the mice chase the farmer's wife across the screen several times:

```
: CHASE 20 0 DO SHTR
           LOOP ;
```

How about making up some words and new characters to show the farmer's wife turning round, carving knife in hand, and chasing the mice back across the screen?

Some machines spoil this kind of graphics display by printing the

cursor after every character. It is useful to have a word to turn off the cursor. This word is for the BBC Microcomputer:

```
: OFF 0 0 0 0 0 0 1 23 8 0 DO >VDU
                              LOOP ;
```

The easiest way of turning it back on again is to change Mode.

The chase scene is the simplest way of animating objects. It works reasonably well for mice, as the scale on which they are displayed is too small for us to expect to see their legs actually moving. The running wife lacks realism, because we ought to be able to see her legs move as she runs. She appears to skate across the screen. The next section shows how to improve the animation.

Leapfrog

Figure 6.8 shows a series of designs for a leaping frog. Design 0 shows it sitting, ready to leap. When it leaps, we need two graphics characters side by side (1 and 2) to show its extended legs and body. Design 3 shows it landing on its forelegs, its rear legs curling up ready to regain the sitting position. This general approach can be used to analyse the locomotion of any kind of animal from a walking person to a bounding leopard. It helps to set out the designs in their relative positions, as in Fig. 6.8, so that the flow of motion is clearly seen.

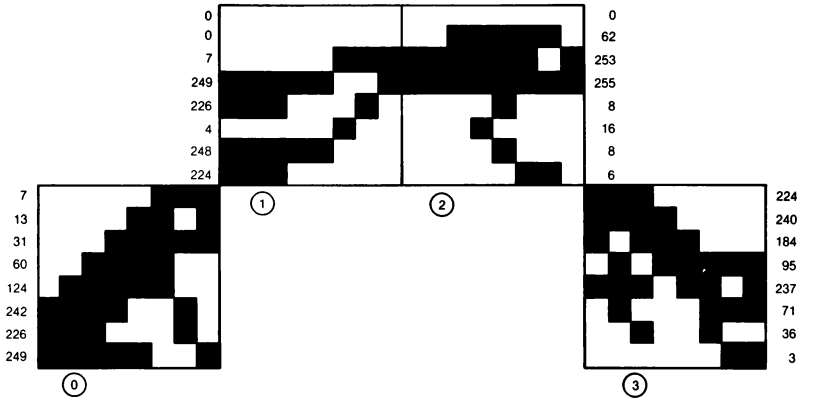


Fig. 6.8. Designs for LEAPFROG.

The frog moves three squares forward at each leap. Each of characters 0 to 2 is displayed and blanked in turn. Then

picture 3 is displayed so that it lands three squares ahead of the square from which it took off. Character 3 is not blanked, but is replaced by character 0, ready for the next leap.

To prepare for the leaps, the values of X and Y for each character are stored in CHARS. We cannot use STORPLACE as we did with the three blind mice, because this would display all four characters immediately. We want only character 0. The solution is to set up CHARS with the values of X and Y that each character *would* have, if it was just about to leap on to the screen from the left:

Character	X	Y
0	−3	15
1	−2	14
2	−1	14
3	0	15

When animation begins, each value of X is to be incremented by 3 to place each character at its required starting position. The values of Y assume that Y increases downward on the screen. If your micro counts rows from the bottom upward, exchange the values of Y given above.

This word puts the required starting values into CHARS (see Fig. 6.9).

```
: READY -3 0 CHARS !  15 1 CHARS !
        -2 2 CHARS !  14 3 CHARS !
        -1 4 CHARS !  14 5 CHARS !
         0 6 CHARS !  15 7 CHARS ! ;
```

Fig. 6.9.

Next we need a delay to give us time to see what is happening:

```
: DELAY 5000 0 DO LOOP ;
```

The action of the frog is shown in Fig. 6.10. LEAP displays the sitting frog, after having 'moved' it 3 squares to the right from its 'starting position' off screen. After a delay, BLANKIT clears the display of the sitting frog. Immediately, MOVEIT is used twice to

```
: LEAP 0 3 0 MOVEIT DELAY 0 BLANKIT
       1 3 0 MOVEIT 2 3 0 MOVEIT DELAY
       1 BLANKIT    2 BLANKIT
       3 3 0 MOVEIT DELAY 3 BLANKIT ;
```

Fig. 6.10.

display the leaping frog characters (1 and 2). Then comes a delay. LEAP then uses BLANKIT twice to clear away the leaping frog, and uses MOVEIT to display the landing frog. After a delay, this is blanked out.

The word HOPS sets the frog leaping across the screen:

```
: HOPS READY 12 0 DO LEAP
                     LOOP 0 3 0 MOVEIT CR ;
```

READY sets up CHARS for the start. Then comes a loop to repeat LEAP 12 times. After this the sitting frog is displayed once more in the final landing position. The CR at the end of the word is to make the computer display the 'OK' message on the next line instead of to the right of the frog.

Happy landings!

To summarise

In this chapter you have found out how to:

- Define a defining word, using CREATE ... DOES>.
- Define an array to hold numbers.
- Define a graphics character, using a FORTH word.
- Make the computer display the character anywhere on the screen.
- Make the computer move the character across the screen, or replace it by another character to give animation.
- Set about writing a FORTH word definition.
- Use DO ... LOOP to make an application run slowly.

You have used these FORTH words:

- CREATE and DOES> to define new defining words.
- ALLOT which we use to reserve n bytes of memory when defining arrays, etc. (n ...)
- PICK and other stack operators (see Fig. 6.4).
- 1+ 'one-plus' (n ... n+1)
- FORTH to ensure that all words used are those of the standard FORTH vocabulary.

Explore more

(1) Create an array to hold 100 values. Write a word using DO ... LOOP to fill the array with the values 1 to 100, in order. Write a

word to display the contents of the array, in order.

(2) Follow up the many suggestions made in the text for devising animated graphics displays.

(3) Write words to produce a display of (a) a walking or running figure, (b) a horse jumping over a fence, (c) a parachutist descending, (d) a clown juggling balls from hand to hand, (e) a car racing around a rectangular track (as seen from above).

Chapter Seven
Interactive FORTH

All the applications used so far were started by pressing the RETURN key. Then the application did whatever it was supposed to do. When it finished, the 'OK' message appeared on the screen. While the application was running, there was no way in which you could have any affect on the computer, other than by pressing ESCAPE, or by BREAK, or switching it off. Drastic actions such as these are not usually described as *interactive*! In this chapter we shall look at ways of interacting with the computer while an application is running without bringing it to a grinding halt.

One of the most useful words provided in FORTH is KEY. It causes the computer to wait until a KEY is pressed. Then it places the ASCII code of that KEY on the stack. Try it:

```
KEY .
54 OK
```

In the example shown above, the '6' KEY was pressed. The ASCII code for 6 is 54 and this was placed on the stack. The ASCII codes for other characters are listed in Appendix B. KEY is one of the most useful words for making the computer wait for input from the user. After the key has been pressed, the number at top-of-stack tells the computer which key it was. The computer can act accordingly, as will be explained in Chapter Eight.

In some applications it does not matter which key is pressed. We simply want to make the computer wait until we are ready for it to continue. The following line demonstrates how to do this:

```
KEY DROP ." HELLO" CR
HELLO
OK
```

The computer waits until a key (*any* key) is pressed, then displays the message.

Some versions of FORTH have the word 'query-KEY', which is similar to KEY except that, the computer continues if a key is not pressed within a given period of time:

```
500 ?KEY
OK
```

The number put on the stack before 'query-KEY' is used tells the micro how many hundredths of a seconds to wait. In the example above, the 'OK' appears as soon as you press a key. If you do not press a key, it appears after five seconds.

Another related word available in some versions of FORTH is INKEY. This makes the computer test the keyboard to find out if a KEY is being pressed, but it does not wait for a key-press. It has the same action as 'query-KEY' would have if there was zero on top-of-stack. If you have INKEY in your version of FORTH, but do not have KEY or 'query-KEY', Appendix A shows how to define them by using INKEY.

String inputs

Single key-presses are very useful but often we want to be able to input a whole string of characters. For example, the application might need to know the user's name. An accounts application might require the user to type in a string of figures. In the rest of this chapter we shall look at a few of the simpler ways in which FORTH can handle strings. In doing this we shall explore a few more aspects of the way the language works.

The word which FORTH uses for string input is QUERY. This makes the computer wait for one or more characters to be typed in. As each character is typed, it is stored in a part of memory called the terminal (or keyboard) input buffer. This continues until the RETURN key is pressed, indicating the end of the string, or until the input buffer is full. The maximum number of characters that can be typed in at one time depends on the version of FORTH. Here is how QUERY works:

```
: NAME? ." TYPE YOUR NAME: " QUERY .S ;
```

This definition has many uses in applications in which the user's name is required. We are taking this as an example which you can easily modify to handle other kinds of string input, such as telephone

numbers, answers to questions, or numerical data contained in a string.

Since we are exploring the string handling of FORTH, the last word of the definition is 'dot-S', to display the stack. The action of NAME? is:

```
NAME?
TYPE YOUR NAME: GUY FAWKES
EMPTY GUY  ?
```

First of all, the message is displayed. The computer then waits until something is typed in. In this example the name 'GUY FAWKES' was typed, followed by RETURN. The response in the third row shows the stack to be empty. It also shows that the micro started to look at what had been put into its input buffer and did not understand it! It searched its list of words and could not find 'GUY' among them.

Obviously, QUERY leaves nothing on the stack. The 'GUY FAWKES' string seems to have disappeared into some inner region of the computer. We need a word which will take the string from the input buffer, and store it in some accessible place. The word used to do this is WORD. WORD takes a string from the input buffer and transfers it to some other part of memory. Exactly where it places it depends on the version of FORTH. In Acornsoft FORTH it places it in the *word buffer*. You can find the starting address of this stretch of memory by using the word 'word-buffer' (WBFR) to place the address of the first byte of the buffer on the stack:

```
WBFR .
1088 OK
```

Here is NAME? re-defined to include WORD;

```
: NAME? ." TYPE YOUR NAME: " QUERY
        13 WORD .S ;
```

And here it is in action.

```
NAME?
TYPE YOUR NAME: KING CANUTE
1088 OK
```

The figure '1088' tells us where the word buffer begins. The word WORD is capable of taking the whole input buffer and transferring it to the word buffer. On most occasions we do not want the *whole*

buffer to be transferred, for we may have typed in only a single short word. To tell WORD where to stop, we have to indicate which character is to be regarded as the last one in the string. A character which indicates the end or limit of a string is called a *delimiter*. WORD expects to find the ASCII code for the delimiter on the stack. Since the string is finished with a carriage return (pressing RETURN), we place the ASCII code for this (13) on the stack before using WORD. A carriage return does not produce a visible character on the screen, but it is interpreted by the computer as an instruction. It is possible to use other characters as delimiters. We have already used the word 'dot-quote' which has " as its delimiter. In the example below, we are using the space (ASCII code 32) as delimiter:

```
OK
: NAME? ." TYPE YOUR NAME: " QUERY
        32 WORD .S ;

NAME?
TYPE YOUR NAME: KING CANUTE
1088 CANUTE  ?
```

Now WORD reads the input string only as far as the first space. The word 'KING' is transferred to the word buffer, but 'CANUTE' is left behind in the input buffer to puzzle the micro! Using a space as a delimiter is a convenient way of ensuring that only one word is taken in at a time. It is also possible to use a letter such as an 'A' as delimiter:

```
: NAME? ." TYPE YOUR NAME: " QUERY
        65 WORD .S ;

NAME?
TYPE YOUR NAME: KING CANUTE
1088 NUTE  ?
```

The string is transferred as far as the first 'A', leaving only 'NUTE' to mystify the micro.

The result of using NAME showed that WORD leaves the address of the start of the word buffer on the stack. WORD has transferred the string to the word buffer, and has told us where to find it.

The usual way to fetch a stored number and put it on the stack is to use the word 'fetch' (@). But it has been explained that FORTH uses *two* bytes to store each number. 'Fetch' takes the address of the first of these two bytes, then gets the values stored in the first byte and the

next one, and uses them to calculate the stored number. In the case of the word buffer, we might expect that each character is stored in a *single* byte. To fetch a single byte and put it on the stack we use the word 'C-fetch' (C@). This is one of a range of FORTH words used for single-byte operations, all of which have a C as part of the word. Here we use 'C-fetch' to find out what is stored in the first byte of the word buffer:

```
1088 C@ .
2 OK
```

If your FORTH includes WBFR, you can use the line 'WBFR C@ .' instead.

The result of this operation is the value 2. This is the number of characters that WORD has put into the buffer. This 2 has nothing to do with 'KING CANUTE'. Since using NAME? we have typed in the direct command given in the line above. The word most recently placed in the word buffer is C@, which is 2 characters long. This reminds us that words such as QUERY and WORD are being used by FORTH all the time to receive input from the keyboard. If we type in direct commands to find out how the buffers are operated, the commands themselves interfere with what we are trying to find out. The way round this is to put the test routines into words, such as NAME?. These are then executed without any further input from the keyboard except that which we want to test. This is why we are forced to define and redefine NAME? so many times in this chapter.

Here is yet another version of NAME?. This one uses 'C-fetch' to fetch the first byte from the word buffer immediately after WORD has dealt with the input string:

```
: NAME? ." TYPE YOUR NAME: " QUERY
        13 WORD C@ . ;

NAME?
TYPE YOUR NAME: ETHELRED THE UNREADY
20 OK

NAME?
TYPE YOUR NAME: MERLIN
6 OK
```

These tests confirm that the number found in the first byte of the word buffer is the number of characters in the string, including spaces if any.

Presumably the next bytes in the buffer contain the ASCII codes for the characters in the string. There is a word COUNT which takes us one step further. Given the start address of the word buffer, left at top-of-stack by WORD, it finds out the number of characters and the start address of the string, leaving these at second-on-stack and top-of-stack respectively. Here we see it in action:

```
: NAME? ." TYPE YOUR NAME: " QUERY
        13 WORD COUNT . . ;

NAME?
TYPE YOUR NAME: MACHIAVELLI
11 1089 OK
```

Count Machiavelli has 11 letters in his name, and it is stored from address 1089 (the second byte of the buffer) onward. To find out how the string is stored, all that needs to be done is to fetch and display these bytes, in order. Figure 7.1 is a version of NAME? which does this. This is the same as the previous definition, as far as COUNT. After this, the stack is operated on as follows (top-of-stack on right)

COUNT leaves	1089	11
OVER	1089	11 1089
+	1089	1100
SWAP	1100	1089

```
: NAME? ." TYPE YOUR NAME: " QUERY
        13 WORD COUNT OVER + SWAP
        DO I C@ .
        LOOP ;
```

Fig. 7.1.

We now have the address of the start of the string at top-of-stack, and the address of the end of the string at second-on-stack. Then comes a DO ... LOOP to fetch and display the contents of each of these bytes, in order:

```
NAME?
TYPE YOUR NAME: CASSANDRA
67 65 83 83 65 78 68 82 65 OK
```

A table of ASCII codes (see Appendix B) shows that these values are the codes for the letters of words typed in.

To sum up, there are three words important for string inputs:

QUERY waits for input from the keyboard.
WORD transfers this to the word buffer.
COUNT tells us how many characters are in the string and where the stored string is to be found.

Creating strings

Some versions of NAME? displayed the name on the screen. This may have been interesting, in that it helped us find out how FORTH deals with strings, but it is not particularly useful. If you have just typed in your name, there is not much point in having the computer display it to you again. What is needed is a way of storing the string away, so that it can be recalled later in the application. In a BASIC program we do a similar thing when we say:

```
INPUT "TYPE YOUR NAME: ";A$
```

What we need to devise is a way of creating named *string variables*, similar to A$, that can be used later in an application as many times as we want.

Since FORTH does not already possess string variables, it is necessary to define a word for defining them. We have already used CREATE and DOES> for defining defining words. This was in Chapter Six when we defined the word ARRAY, which was then used to define named arrays, such as CHARS. Here is another chance to see how CREATE and DOES> operate. Their action is in two parts:

(1) What is to be done by the defining word when it defines, and
(2) What is to be done by the word it defines, when it is used in an application.

When it defines, the defining word has to allocate a piece of memory to contain the name of the string (for example, NAME$, DATE$) and the ASCII codes of the characters in the string. It then has to transfer the codes of the string from the word buffer to the space it has just allocated. One small difference from BASIC is that although it is a good idea to let the names of strings end in '$', to show that they are strings, this is not essential. In FORTH we can

give strings any kind of name we like.

When a word has been defined as a string, it consists of a part containing its name and a part containing the string itself. When the name of the string is used in an application, its action should be to make the computer display the string on the screen. This gives us the equivalent of the BASIC statement 'PRINT A$'. Before we can see exactly how to do this, we need to look more closely at how words are defined and at where the definitions are stored. After we have done this we will return to the subject of defining strings.

New words

Switch off the computer, then reload FORTH. Or, if your version provides a method of doing so (such as typing COLD), perform a cold start. This clears the stack and, more important for this exploration, clears away all the word definitions which you have made. You begin with only those words provided by the writers of your version of FORTH. Now type VLIST. You have probably done this many times already for it is probably the first thing you ever did. Writers of FORTH manuals for beginners are fond of putting this first in the manual. It's the easiest way for a beginner to get something impressive on the screen! Just to be different, we have left VLIST until Chapter Seven, the first chapter in which we need to use it.

What you see on the screen is a list of all the words your FORTH contains. You will now able to spot 'colon', QUERY, +, WORD, 'dot-quote' and all the other words you have used, apart from those you have defined yourself. In the usual back-to-front method of FORTH, the list is in reverse order. The first word the writers defined is listed last, followed by the inevitable 'OK'. The word most recently defined is at the top of the list. The words are stored in memory in the order shown by VLIST. Early words are low down in memory, the more recent the word the higher in memory is its address. These stored words constitute the *dictionary* that FORTH uses when it operates. Any new words that you define are stored in memory just above the existing dictionary. Their names are added to the end (the top) of the list. The word HERE tells us the address of the first byte available for receiving the definition of a new word:

```
HERE .
15461 OK
```

The value obtained depends on the version of FORTH. If the value you obtain differs, as well it may, then make the necessary allowances in the description which follows.

We are just about to undertake an exploration of the FORTH dictionary. The idea of this is to help you understand how FORTH works in your computer. Knowing about this is helpful, but not essential to programming in FORTH. If you would rather get on with the programming, pass by the remainder of this section and continue with the next section, *Back to strings.*

As an example of how words are stored, let us define a simple word:

```
: DOUBLE DUP + . ;
```

Given a number on the stack, DOUBLE displays double its value. What effect has this definition had on the value of HERE?

```
HERE .
15480 OK
```

HERE has moved upward in memory by 19 bytes. This shows that the definition of DOUBLE occupies 19 locations from 15461 to 15479 in memory. To see what bytes these locations hold, we define SHOW:

```
: SHOW DO I C@ . LOOP ;
```

Because SHOW contains 'C-fetch' it fetches the values of the *bytes* held in each location. All we need to do is to put the end (plus 1) and beginning of DOUBLE on the stack and use SHOW:

```
15480 15461 SHOW
134 68 79 85 66 76 197 190 59 179
32 49 31 176 30 140 47 203 29 OK
```

What do all these numbers mean? Let us take them a few at a time:

134 subtract 128 from this one: the result is 6 which is the number of letters in the name of the word (DOUBLE).

68, 79, 85, 66, and 76 are the ASCII codes for the letters D, O, U, B, and L.

197 subtract 128 from this one too, giving 69, the ASCII code for E, the last letter of the name.

The remaining numbers do not make much sense until we realise

that these are values being stored as *double-bytes*. To find out what these values are we use 2SHOW:

```
: 2SHOW DO I @ . 2 +LOOP ;
```

This loop uses 'fetch' instead of 'C-fetch' so it fetches numbers as *double bytes*. It also uses the word +LOOP. This gives the equivalent of STEP in a FOR...NEXT loop in BASIC. Here, the 2 before +LOOP makes the loop index increase in steps of 2, so that each 'fetch' reads the next double-byte. The double-byte section of DOUBLE begins at address 15468:

```
15480 15468 2SHOW
15294 8371 7985 7856 12172 7627 OK
```

The numbers are interpreted as follows:

15294 This is the address of the next word in the dictionary. This value is called the *Link Field Address*.

8371 This is the first address of the machine code which has been compiled to execute DOUBLE. This value is called the *Code Field Address*.

7985 to 7627 These are values needed by the machine code routine. This collection of values is called the *Parameter Field*. The address of the first of these (15472) is called the *Parameter Field Address*.

Let us look more closely at the values in the parameter field. We might guess that these are the addresses of various routines used in executing the word. In particular they might be the addresses of the word DUP, 'plus' and 'dot'. This can be checked on if your FORTH has the word 'tick'. This word causes the parameter field address of any word to be put on the stack. In Fig. 7.2 it is used to find the PFA's of the words used in defining DOUBLE. The last word

```
' DUP .
7987 OK
' + .
7858 OK
' C@ .
8231 OK
' . .
12174 OK
' EXIT .
7629 OK
```

Fig. 7.2.

'ticked' is EXIT. This was not part of our definition, but is included automatically to terminate each word defined by 'colon'. The figures obtained by 'tick' each differ by 2 from the corresponding values displayed by 2SHOW. 'Tick' is giving the parameter field address of each word, while 2SHOW is giving the code field addresses which appear in the parameter field of DOUBLE.

To sum up, a word definition consists of the following parts:

The Head: containing:

The Name Field – one byte gives the number of letters (+128), and the others give the ASCII codes (+128 on the last one).

The Link Field – two bytes giving the address of the next-door word in the dictionary.

The Code Field – two bytes giving the address of the machine code which executes the word.

The Body: holding the Parameter Field, in which are the addresses or other values needed by the machine code routine.

When FORTH is told to execute a word, it first searches its dictionary for that word. It begins with the most recent word and checks its name in the Name Field. If this is not the word it wants, it looks in the Link Field to find the address of the next word. Then it goes to this word to see if it is the one it wants. It works its way through the dictionary until it finds the word it wants. Then the Code Field tells it where to find the machine code routine. This routine calls upon other words, the addresses of which are to be found in the parameter field. These words themselves have their own machine code routines and their own parameter fields, which in turn call upon other words. In the primitive words of FORTH the address in the parameter fields are those of further machine code routines.

It might be thought that the time required for FORTH to search its dictionary for a given word would make FORTH a very slow language indeed. However, when we define a new word, this goes at the *top* of the dictionary. As we build up a more complex application from existing words the new words are all stored close to the top of the dictionary. Quite often the whole of an application is called on by typing a single word, which is likely to be the topmost word of all. If this is so, FORTH finds the word it wants at the top of the dictionary. Its search is ended immediately!

There is no need for it to go searching through its dictionary when the application is running. All such searching was done as each new

word was compiled. At run time, the parameter field of each word contains the addresses of the machine codes routines or the code field addresses of each word it calls upon. FORTH *jumps directly* from one address to the next as it executes each word, performing first this one piece of machine code, then the next. This is the secret of the way FORTH combines high speed with great flexibility.

With VLIST as your chart and with SHOW and 2SHOW as your instruments, you are ready to explore the deepest regions of the FORTH jungle. Starting from a word such as DOUBLE, we have been led to the words used in its definition. Next you could explore the definitions of these other words. Gradually you could trace your way up the tributaries, the words in defining these words, until you come to the primitives, which use machine code directly. You are led to the starting addresses of these routines, and, if you understand machine code, can get right to the sources of FORTH. In following these threaded pathways you are travelling the same routes that FORTH uses when it executes an application. Few may wish to venture so far, but the way is open for those of an exploring nature.

Before we return to define words to define string variables, we will look at the definition of the word used for numeric variables, VARIABLE. First to get our bearings:

```
HERE .
15529 OK
```

Next to define a variable:

```
VARIABLE WEATHER
```

How much space does WEATHER occupy?

```
HERE .
15543 OK
```

It extends from 15529 to 15542, so we use SHOW to take a closer look:

```
15543 15529 SHOW
135 87 69 65 84 72 69 210 143 60 12 33 0 0 OK
```

Taking 128 from 135 gives 7, the number of letters in the name WEATHER. The table of ASCII codes confirms that the numbers 87 to 210 are the name in code (subtract 128 from 210 to get 82, the code for R). The last six numbers are the double bytes of the link field, the code field and the parameter field, respectively, one double byte for each. We use 2SHOW to read them, starting from address 15537

(count your way along the row of single bytes from 15529 to find the starting address):

```
15543 15537 2SHOW
15503 8460 0 OK
```

The link field address is 15503. The code field address is 8460. Could the zero be the stored value of WEATHER? We have said that VARIABLE initialises each variable to zero when it is defined, so this seems likely. To check that this is so, we set WEATHER to a particular value:

```
123 WEATHER !
OK
```

Now try 2SHOW again:

```
15543 15537 2SHOW
15503 8460 123 OK
```

And there it is, stored in the last two bytes of the parameter field. Counting along the byte from 15529 tells us that the addresses at which the variable is stored are 15541 and 15543. When we first discussed the action of VARIABLE it was stated that it puts on the stack the address at which the value is stored. To confirm this, try:

```
WEATHER .
15541 OK
```

Which shows that the value put on the stack by VARIABLE is indeed the address we expected. It is the first address of its parameter field. Using 'fetch' after WEATHER will bring the value of the double byte (15541–15542) on to the top-of-stack.

Back to strings

The next task is to define the defining word STRVAR which will define string variables. This is really a simple matter, for all that the string variable needs to have is a head containing its name and the usual link field and code field addresses, plus a body consisting of enough bytes to hold the string to be stored there. Here is its definition:

```
: STRVAR CREATE ALLOT ;
```

STRVAR expects to find a number on the stack to indicate how many bytes are to be set aside for the string. The number is to be one greater than the maximum length of string, to allow an extra byte in which to store the string length. There is no DOES> section of this definition, for there is nothing special for a string variable to do when used. When the string variable is used, it *automatically* puts its parameter field address on the stack, and this is all we need. In this respect STRVAR acts in the same way as VARIABLE. Now we are ready to define a string variable, for example:

```
16 STRVAR NAME$
```

The line above defines a string variable called NAME$, which can hold a string consisting of up to 15 characters. At this point you could check that STRVAR has worked. Use the technique described in the previous section. First find the value of HERE. Then use SHOW to display 26 bytes up to the one before HERE. The first should hold 5 (the number of characters in NAME$). Next come 5 bytes holding the ASCII codes for NAME$, then there are 4 bytes for the link field and code field addresses. Then there are 16 bytes for the parameter field. These may hold any values at present, for the definition of the word does not provide for clearing the parameter field.

Next we need a word for storing a word in NAME$, or in any other string. A good name for this word is STR!, the '!' reminding us of the word used for storing numerical values. We intend to use STR! after WORD, and the name of the string variable, as in 'WORD NAME$ STR!'. WORD puts the start of the word buffer (WBFR) on the stack. The string variable puts its PFA (parameter field address) on the stack. STR! will operate on these two values. Here is its definition:

```
: STR! OVER C@ 1+ CMOVE ;
```

Let us see how this works on the stack (top to right):

STR! finds this		WBFR	PFA	
then does	OVER	WBFR	PFA	WBFR
	C@	WBFR	PFA	count
	1+	WBFR	PFA	count+1

'C-fetch' fetches the first value from the word buffer, which, as explained earlier, is the number of characters in the string. This is incremented by 1 so that the string variable will receive not only the string but the number of characters as well.

The final word used in STR! is CMOVE. This word transfers bytes from one part of memory to another. It expects to find on the stack:

from\to\count

'From' is the starting address of the block of memory the bytes are to be transferred from. In this case this address is to be the start of the word buffer (WBFR). 'To' is the starting address of the block of memory the bytes are to be transferred to. In this case this is to be the beginning of the parameter field (PFA) of the string variable. 'Count' is the number of bytes to be transferred. The list above shows that the stack has everything ready for CMOVE to operate on. The bytes are transferred from the word buffer to the parameter field.

We also need a word for taking the string which is stored in a string variable and displaying it on the screen. One of the simplest possible of such words is .STR:

```
: .STR COUNT TYPE CR ;
```

In use, this is to be preceded by the name of the string, as in 'NAME$ .STR'

The name of the string puts the PFA on the stack. COUNT can operate on this, just as it did with WORD. It leaves the address of the beginning of the string (PFA+1) at second-on-stack and the count at top-of-stack. These two values are those required by the word TYPE. This transfers the given number of bytes to the screen or other output device. The CR is optional, but is useful in a demonstration such as this, for it makes the computer print its 'OK' message on the next screen line.

Now for the ultimate definition of NAME?:

```
: NAME? ." TYPE YOUR NAME: "
        QUERY 13 WORD
        NAME$ STR! ;
```

This displays its customary message and waits for the user to type in the name. When RETURN is pressed, the typed string is transferred from the input buffer to the word buffer and from there to the parameter field of NAME$. There it stays unless altered at some later stage of the application by using NAME$ a second time. Here it is in action:

```
NAME?
TYPE YOUR NAME: TARAS BULBA
OK
```

At some other stage in an application we may want the stored name to be displayed. This is one way of displaying it:

```
NAME$ .STR
TARAS BULBA
OK
```

Usually the format would be more elaborate. In a game for example, we could display the name of the person whose turn it is to play:

```
: TURN1 ." IT IS " NAME$ .STR
        ." 'S TURN" CR 7 EMIT ;

 TURN1
 IT IS TARAS BULBA'S TURN
 OK
```

A similar word, TURN2 would be used to display the other player's name. The word .STR was redefined, omitting the CR, for use in TURN1. TURN1 uses yet another FORTH word, EMIT, that causes a character to be sent to the output device. EMIT requires the ASCII code of the character to be on the stack. In this example the code was 7, the ASCII code for BEL. The usual action of this is to make the micro produce a 'beep' from its loudspeaker. If your micro does not recognise this character, maybe it has a BEEP word that you can substitute for '7 EMIT'.

The PFA put on the stack by a string variable (created by STRVAR) and the value (WBFR) left on the stack by WORD are much alike. Both point to an address which contains the number of characters in the string and in both cases the string begins at the next address. This makes it very easy to transfer strings from one string variable to another. Having stored 'TARAS BULBA' in NAME$, as above, we can define another string variable NEXT$, using STRVAR. We can transfer 'TARAS BULBA' directly to NEXT$ by:

```
NAME$ NEXT$ STR!
OK
```

And this shows that it has worked:

```
NEXT$ .STR
TARAS BULBA
OK
```

If you are intending to write applications using strings, it is worth

while saving the words we have devised in this chapter. Figure 7.3 shows a complete listing.

```
SCR # 17        11 H
  0 ( STRING VARIABLES )
  1 : SHOW DO I C@ . LOOP ;
  2 : 2SHOW DO I @ . 2 +LOOP ;
  3 : STRVAR CREATE ALLOT ;
  4 : STR! OVER C@ 1+ CMOVE ;
  5 : .STR COUNT TYPE CR ;
  6 : NAME? ." TYPE YOUR NAME: "
  7           QUERY 13 WORD
  8           NAME$ STR! ;
  9
 10
 11
 12
 13
 14
 15
```

Fig. 7.3.

There is only one further point to be made. The words have no error-checking facilities built in to them. For example, if NAME$ is defined so as to allow up to 15 characters in the string and if the user types in a name longer than this, only the first 15 characters will be displayed. Maybe this does not matter in a given application. Too long a name may spoil the display, so persons with over-long names will have to put up with a truncated version. We shall return to the subject of error checking in Chapter Eight.

To summarise

In this chapter you have found out how to:

- Make applications more interactive.
- Deal with string inputs.
- Create and store string variables.
- Explore the FORTH dictionary.

You have used these FORTH words:

- The 'key' words, KEY, ?KEY and INKEY which detect key-presses (...ASCII code).

- QUERY accepts input of 1 or more characters from the keyboard, storing their ASCII codes in the input buffer.
- WORD transfers characters from the input buffer to the word buffer (or to HERE in some computers), until it finds a delimiter. Puts the total number of characters in the first byte of the word buffer (ASCII code of delimiter ...).
- COUNT gives address of the start of word buffer, its stack action is (address ... address of start of string\n).
- EMIT displays a single character (or sends it to some other output device currently in use) (ASCII code ...).
- TYPE displays one or more characters (or sends them to some other output device currently in use) (address of start of string\ count ...).
- VLIST lists the words in the FORTH vocabulary (...).
- HERE (... address of first unused byte in the dictionary).
- C@ 'C-fetch' puts contents of a byte on the stack (address ... contents).
- CMOVE moves a block of bytes (containing 'count' bytes, starting at address 'from') to another part of memory (starting at address 'to') (from\to\count ...). 'From' area and 'to' area should not overlap.
- +LOOP used instead of LOOP when the loop index is to be stepped by any value other than +1 (step ...).
- ' 'tick' when used as a direct command from the keyboard, as in "' SWAP ", it places the parameter field address of the word on the stack (... PFA).

You have learned:

- That a FORTH word consists of:
  A Head containing the Name, Link and Code Fields.
  A Body containing the Parameter Field.
- The Parameter Field contains addresses and other data needed when the word is executed. It is used for storing values when the word is a variable, constant, array or string variable.

Explore more

(1) Define a new version of the defining word for string variables, STRVAR, which fills each string variable with spaces when each is defined.

(2) Define a word HALF which displays half the value of any number placed on the stack. Then follow up the definition as far as you can, threading your way through memory, using the techniques described in this chapter.

(3) Follow up the definition of a word such as 2/ or SWAP.

Chapter Eight
Taking Decisions

One of the most common steps in a program or application is to ask the user a question which requires the answer 'Yes' or 'No'. Examples are:

'Do you want to play again? (Y/N)'
'Is the tape recorder ready? (Y/N)'
'Do you require printout? (Y/N)'

Note that each of these questions is followed by "(Y/N)" to make it clear to the user that the required response is either Y or N. The user presses one key or the other. Then the computer has to decide which key was pressed. If Y was pressed, it acts in one way, if N was pressed, it acts in a different way. The program or application branches at this point according to the response of the user.

The words used for taking a decision are IF and THEN. These words are used in BASIC too, but in a rather different way, as we shall see. In FORTH, they must always be used inside a 'colon' definition, so let us define a word to use them:

```
: YN? KEY 89 = IF ." YES"
                  THEN CR ;
```

YN? is a word to find out if the user has pressed Y or N. It could be preceded by a message displayed by 'dot-quote' to ask the question. YN? uses KEY to wait for a single key-press, as explained at the beginning of Chapter Seven. KEY leaves the ASCII code of the key on the stack. Then 89 is put on the stack, so there are now two numbers on the stack. Next comes the word 'equals'. This is different from the words we have met so far. It is a *relational operator*. It reports on the relationship between the top two values on the stack. In doing this, both values are removed from the stack and are replaced by a value known as a *flag*. If the two values were equal, the flag is 1. If they are unequal, the flag is 0. The action of 'equals' is

easy to investigate (See Fig. 8.1). Equals leaves 1, only when the numbers are equal. In YN?, Key should leave either 89 or 78 on the stack, depending upon whether the Y or the N key was pressed. YN? places 89 on the stack and then uses 'equals' to find

```
4 4 = .
1 OK

4 5 = .
0 OK

5 4 = .
0 OK
```

Fig. 8.1.

out if the number left by KEY equals 89 or not. In other words, was the Y key pressed? If it was pressed, the top-of-stack will be 1, if the N (or any other) key was pressed it will be zero.

Next in the definition of YN? comes IF. Like most other FORTH words, it expects to find a value on the stack. In particular, it expects to find a 0 or a 1. It interprets a 1 to mean 'true'. That is to say: "It is true that the user pressed the Y key". It interprets a 0 to mean 'false', or 'not true' – the user did not press the Y. IF is a decision-taking word.

Depending on the results of the test, true or false, the computer is sent in one of two directions. If the result is true, the words following IF, are executed, as far as THEN. After this the computer continues executing the words, if any, which follow THEN. If the result is not *true*, the words between IF and THEN are not executed. The computer jumps straight to THEN and continues from there.

In the operation of the word YN?, after the Y key has been pressed, the computer displays YES, then jumps to the carriage return after THEN and executes that:

```
YN?
YES
OK
```

If N (or some other key) is pressed, the computer jumps straight to THEN and continues to the carriage return without printing YES :

```
YN?
OK
```

To sum up, the form of a conditional branch is:

<FLAG> IF <action if FLAG is true> THEN <continue, whether true or false>

Relational operators, such as 'equals' place either 0 or 1 on the stack, but IF also interprets *any* non-zero value as true:

```
: TRUE? IF ." TRUE" THEN CR ;
```

Use TRUE? after placing various values on the stack. It shows whether IF takes them as being true or not (see Fig. 8.2). The definition of YN? displays "YES" if Y is pressed but does nothing if it

```
7 TRUE?
TRUE
OK

0 TRUE?
OK
```

Fig. 8.2.

is not pressed. We may want the computer to take one action if Y is pressed, but to take an entirely different action if it is not. This requires an extra word, ELSE. The computer branches one of two ways and performs an action either way. The form of such a conditional branch is:

<FLAG> IF <Action 1 if FLAG is true> ELSE <Action 2 if FLAG is false> THEN <continue, whether true or false>

Here is this form of branch applied to YN?:

```
: YN? KEY 89 = IF ." YES"
              ELSE ." NO"
              THEN ."  THANK YOU!" CR ;
```

Figure 8.3 shows what happens. If Y is pressed, it displays "YES". This corresponds to Action 1 above. If N or some other key is

```
YN?
YES THANK YOU!
OK

YN?
NO THANK YOU!
OK
```

Fig. 8.3.

pressed, it displays "NO". This corresponds to Action 2. After that, whatever key is pressed, it displays "THANK YOU!" and executes a carriage return.

YN? is now in a form that can be used in many applications. You could adapt it to other pairs of keyed letters (e.g. U for up, D for down) by replacing the 89 with an appropriate ASCII code. In using YN?, it is assumed that the Y is the more important response. If any letter other than Y is keyed, the computer takes this to be the equivalent of N, by default.

It may be that it is essential that the user types Y or N and that any other response is not to be accepted. In this case we shall have to test the input to see if it is Y. If it is not Y, we shall have to test it *again* to see if it is N. If it is not N either, we shall have to repeat the word until either Y or N is typed. The new definition of YN? looks rather complicated but is easy to understand if taken a bit at a time (see Fig. 8.4).

```
: YN? KEY DUP 89 =
      IF ." YES" DROP
      ELSE 78 =
           IF ." NO"
           ELSE YN?
           THEN
      THEN CR ;
```

Fig. 8.4.

Figure 8.5 is a flow-chart of this version of YN?. The first point is that it uses DUP to duplicate the value placed on the stack by KEY. Then 89 = tests to see if Y was pressed. If so, "YES" is printed and the other value left by KEY is DROPped from the stack.

However, if the test for Y gives a false result we next have to test for N. This comes in the ELSE section of the word. Testing is done by 78 =. Following this is a second IF, to ascertain the result of the test. We have a second branching routine nested inside the first one. If the test gives a true result, showing that N was pressed, "NO" is printed. But what if this test too gives a false result? This inner routine needs an ELSE action to cover such an eventuality. As has been specified above, the computer must wait until the user types in one of the acceptable keys, Y or N. So ELSE is followed by YN?. Before considering this interesting development, note that the definition ends with two THENs, one for the outer branching routine and one for the inner routine. If the key was Y, tested by the outer routine, the computer jumps straight to the outer (last) THEN and leaves the

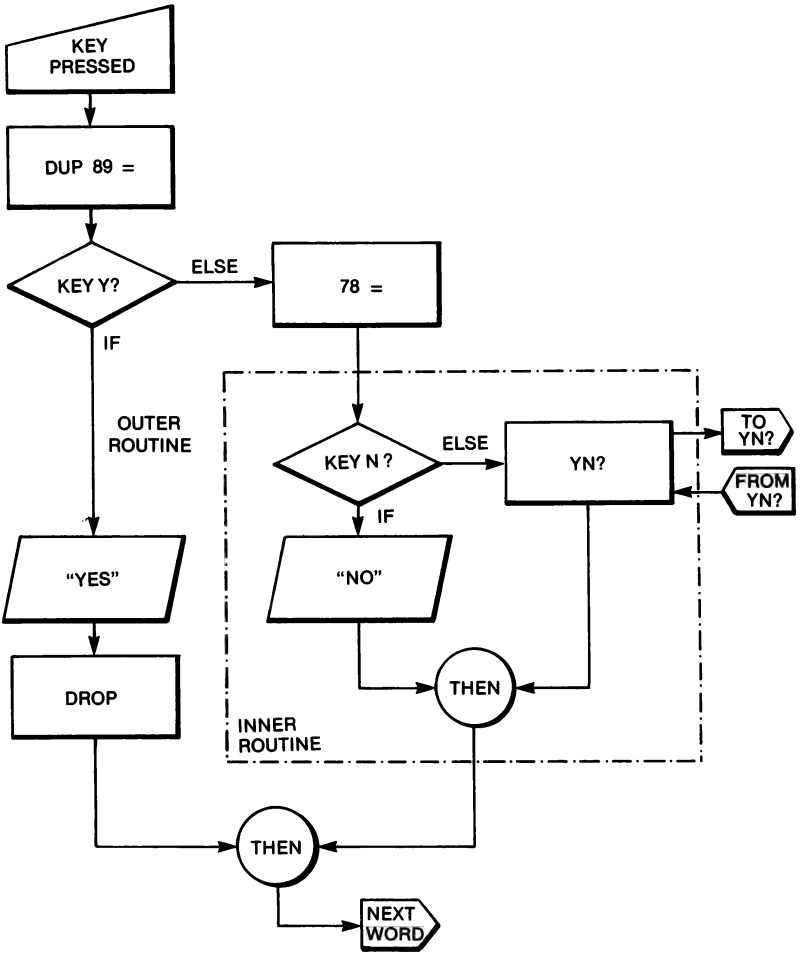


Fig. 8.5. Nested IF ... ELSE ... THEN routines in YN?. The inner routine is nested in the ELSE branch of the outer routine (compare with *Fig. 8.13*).

word. If the key was N it jumps to the first THEN and passes straight on to the second THEN before leaving the word.

Now to return to what happens if the key is neither Y or N. In this event, YN? calls itself. It does this as often as is necessary, until the user has responded acceptably. When a routine calls itself, we refer to it as a *recursive* routine.

In FORTHs such as the Acornsoft FORTH used on the BBC Microcomputer, there are checks in the system to prevent words from being compiled if they contain words that are not already compiled. However, to allow for those who wish to define recursive

words, special defining words 'R-colon' and 'R-semicolon' may be used. These are used in the same way as 'colon' and 'semicolon', but allow you to include the word in its own definition. With other versions of FORTH, such as that of the Jupiter Ace, 'colon' and semicolon' may be used when defining recursive words.

Using recursion in this way may not be thought of highly by some since, each time the word calls itself, it jumps deeper into a nest of IF ... ELSE ... THENs. Sooner or later the micro may not be able to nest the routines to sufficient depth, and an error message will result. On some machines, there is no practical limit. For example, Acornsoft FORTH tolerates at least 30 wrong responses (we gave up trying after this) and will then still accept a correct one. This gives the user the opportunity to key in all letters of the alphabet plus some numbers and symbols as well before arriving at Y or N. The Jupiter Ace allows three invalid entries, accepting a valid entry on the fourth attempt. This may not be quite enough for safety, in which case you can adopt an alternative method, as will be explained in Chapter Nine.

A string too long

One of the dangers of the string handling words defined in the last chapter is that a user may key in a name which is longer than the string variable is intended to hold. We allowed up to 15 characters, which is enough for most people, but there are those with long names who like to type them out in full. READ? accepts up to 80 characters, possibly more in some FORTHs, and STR! will quite happily put these into the parameter field of NAME$ (or any other string variable), even though the parameter field is not long enough to hold them. All seems to go according to plan – you can even use .STR to display the long string – but there is trouble in store. When you try to use the word which was *defined immediately* after NAME$, it seems as if the computer has forgotten all about it. If you VLIST to find out what has happened to it, you may find part of your long string appearing in the list instead of the name of the word you thought was there. Other odd things may appear too! What has happened is that STR! has written the *whole* of the long string into memory and the part that would not fit into the parameter field of NAME$ has been stored in the space previously taken up by the word defined after NAME$. Its name field now contains part of

your long string and, if the string is long enough, its link field and other fields will have been over-written too.

We need a way of checking that the string is not too long to fit into NAME$. What we do when we find that the user has typed in a string too long is a matter of preference. Maybe the computer should emit a loud beep and a warning message should appear. This can be easily arranged but, to keep the discussion simpler, we will just trim off the excess characters and store the shortened string in NAME$. This approach often causes less confusion to the user who should, in any case, have been warned either in the manual or on the screen that only a limited number of characters may be typed in.

The word to trim the string to size is TRIM:

```
: TRIM DUP C@ 7 >
        IF 7 OVER C!
        THEN ;
```

This word is to be included in the definition of NAME? immediately after WORD. In this position it is able to trim the string before it is transferred from the word buffer to NAME$. WORD leaves the address of the word buffer on the stack, so TRIM must do the same, ready for use by STR!. In the meantime it must find out how long the string is and, if it is too long, reduce it to the correct length. What we need to do is to test the value stored in the first byte of the word buffer. If this is greater than the maximum allowed for in READ$, then this value must be reduced to an acceptable value.

TRIM duplicates the address of the word buffer (WBFR), so that it can work one copy, leaving the other for use by STR!. The changes on the stack are:

WORD leaves		WBFR
Then TRIM does	DUP	WBFR WBFR
	C@	WBFR count
	7	WBFR count 7
	>	WBFR flag
	IF	WBFR
	7	WBFR 7
	OVER	WBFR 7 WBFR
	C!	WBFR
	THEN	WBFR

The word *greater than* (>) is another testing word, similar to 'equals'. It puts a flag on the stack, which indicates whether second-on-stack

is greater than top-of-stack or not. If second-on-stack (i.e. count, the number of characters in the word buffer) is greater than top-of-stack (7) then the condition is true and the flag is 1. If count is equal to or less than 7, the flag is 0.

IF operates on the result of the test. If the flag is 0 (false), the computer is sent straight on to THEN. WBFR is unchanged and all is ready for the string to be transferred to NAME$. If the string was too long, the flag is 1 (true). In this event the maximum allowed value (7) is put on the stack, the address of WBFR is copied to top-of-stack and C! is used to store 7 in the first byte of the word buffer. The computer proceeds to THEN with this changed value in WBFR. Later when STR! operates, it reads the string length as 7 and so transfers only the first seven characters to NAME$.

Having defined TRIM (and STRVAR, STR! and .STR, if they are no longer in the dictionary) we redefine NAME$ to hold only 7 characters (plus 1 for the length). The final version of NAME? is shown in Fig. 8.6.

```
8 STRVAR NAME$
OK

: NAME? ." TYPE YOUR NAME: " QUERY
        13 WORD TRIM NAME$ STR! ;
OK
```

Fig. 8.6.

Figure 8.7 shows the new version of NAME? in action. TRIM demonstrates the use of 'greater than' in conditional routines. There is also the word 'less than' (<). This leaves a true flag on the stack if

```
NAME?
TYPE YOUR NAME: FRANCIS
OK
NAME$ .STR
FRANCIS
OK

NAME?
TYPE YOUR NAME: ROGER BACON
OK
NAME$ .STR
ROGER B
OK
```

Fig. 8.7.

second-on-stack is less than top-of-stack. If second-on-stack is equal to or greater than top-of-stack, it leaves a false flag.

Rounding

In Chapter Five the conversion of francs to pounds and pence worked well enough, except that fractions of pence were ignored. For holiday cash transactions the odd fractions of pence are nothing to worry about. But there are other kinds of conversion in which it is essential to round off the figures to the nearest whole integer. In rounding off, the convention is that fractions amounting to less than half of the unit are ignored. If the actual answer is 27.4, for example, we call it 27. Fractions that are equal to or greater than *half* of the unit are rounded up. If the actual answer is 27.5 or 25.8, for example, it is rounded up to 28. We will now do the same thing with the fractions of pence in the conversion application. The same method may be applied to rounding of any other figures.

To begin with, define the variable RATE, and set it to 12. The definitions of POUNDS and PENCE are slightly different from those in Chapter Five (see Fig. 8.8). Both POUNDS and PENCE

```
VARIABLE RATE
OK
12 RATE !
OK
: POUNDS RATE @ /MOD SWAP ;
OK
: PENCE 100 * RATE @ /MOD SWAP ;
OK
```

Fig. 8.8.

contain RATE @ instead of just RATE because RATE is now a VARIABLE instead of a CONSTANT. In PENCE we do the multiplication and division in two stages instead of in one. Before, we used 'times-divide'; now we use 'times' and 'divide-mod'. The latter leaves the result of the division (the whole number of pence) at top-of-stack and the remainder at second-on-stack. The SWAP brings the remainder to top-of-stack so that we can decide whether to round it down or up.

The remainder is what is left after dividing by RATE, which is 12 in this example. If the remainder comes to less than *half* of RATE (less than 6) we ignore it. If it comes to half of RATE or more, we

add 1 to the number of PENCE. Rather than try to base the decision on half of RATE, which would be awkward if RATE was an odd number, such as 13, we double everything up before making the test. This means that the remainder is to be ignored if *twice* the remainder is less than 12, the *full* value of RATE. This is how we test the remainder:

```
: ROUND 2 * RATE @ - 0<NOT + ;
```

The remainder (at top-of-stack) is doubled by 2 *. Then RATE is put on the stack by RATE @. This time we use a more direct method of taking the decision. First of all we use 'subtract' to take RATE from the doubled remainder. If the result of the subtraction is less than zero, we are to round down. If it is equal to or greater than zero, we are to round up. The word 'zero-less' leaves a true flag (1) on the stack if the value at the top-of-stack is less than zero. Conversely, if the value on the stack is zero or more than zero, 'zero-less' leaves a false flag (0) there.

The result of the test is to be interpreted like this:

IF flag is 1 – round down
IF flag is 0 – round up

At this stage we could use IF...ELSE...THEN to take the action required, but there is a quicker way to get the same result. To round up we add 1 to the pence value; to round down we add nothing. We need to add 0 or 1, and the flag is either 0 or 1. But the flag is the 'wrong way round'; it is 1 when we want to add 0, and it is 0 when we want to add 1! Another FORTH word comes to the rescue. This is NOT which, being a negative, turns true into false and false into true. In other words, it changes the flag to its opposite. After NOT the top-of-stack is:

0 when we have to round down
1 when we have to round up.

The value for pence is already at second-on-stack, so it needs only a 'plus' to finish the rounding.

To complete the application we define FORMAT exactly as before, but include ROUND in the definition of EXCHANGE:

```
: FORMAT ." £" 1 .R ." -" . CR ;
: EXCHANGE POUNDS PENCE ROUND SWAP FORMAT ;
```

Converting 58 francs gives £4–83 with 0.33333 pence remainder. This is to be rounded down:

```
58 EXCHANGE
£4-83
OK
```

Converting 59 francs gives £4–91 with 0.66667 pence remainder. This is to be rounded up to £4–92:

```
59 EXCHANGE
£4-92
OK
```

The general principle of this example can be applied to rounding of other integers after division. You need to have a variable (or constant) as the divisor and use the name of this (or its value) in place of RATE in the definition of ROUND.

UFO

Computer games provide many examples of decision taking. The rest of this chapter is devoted to describing a game called UFO, which involves a lot of decision-taking by the micro. It is described in detail to give you plenty of examples of the uses of relational operators and conditional branching. The detailed descriptions will make it easy for you to adapt and extend the game. Many of the words used are designed to be useful in other games applications, so a study of this game will help you begin to design other games of your own.

Rather than break the text by printing out the words individually, they are printed out in screens, which are numbered 18 to 22 (Figs. 8.9, 8.11, 8.12, 8.14 and 8.15). When you type in the game you can, of course, put them on any five consecutive screens you choose.

The game is simple in outline, which means that there is plenty of scope for additions. It begins when the word UFO is typed and RETURN is pressed. The screen clears to blue (if available on your computer) and an unidentified flying object (UFO) appears at the top left corner. Its colour is flashing green and magenta (if available on your computer). It moves diagonally across the screen, 'bouncing off' the edges when it reaches them. Your spacecraft is near the

bottom left corner. Move it across the screen by pressing one of the keys:

Z	left
X	right
;	up
/	down

The keys used can be changed to suit your micro or your own preferences as will be explained later. Your aim is to wait for the UFO to pass by and then 'jump' on it by pressing the correct key at exactly the right moment. If you score a jump, you hear a beep and your score is incremented by 1. However, if by chance you happen to be in the way of the UFO and it runs on to you, you hear a longer beep and your score is decremented by 1. The game runs for several minutes, at the end of which your score is displayed. Naturally, the aim is to get as big a score as possible within the allotted time.

The words are described screen by screen:

PART 1 (Screen 18) (Fig. 8.9)

```
SCR # 18       12 H
  0 ( UFO GAME - PART 1 )
  1 VARIABLE SCORE
  2 VARIABLE MOVEX VARIABLE MOVEY
  3 VARIABLE YOUX  VARIABLE YOUY
  4 : ARRAY CREATE 2 * ALLOT DOES> SWAP 2 * + ;
  5 4 ARRAY CHARS
  6 : CHARDEF 23 >VDU 224 + 9 0 DO >VDU LOOP ;
  7 129 102 24 24 255 219 126 60
  8 0 CHARDEF
  9 153 126 90 255 255 90 126 153
 10 1 CHARDEF
 11 : PLACEIT 31 >VDU SWAP >VDU >VDU
 12 224 + >VDU ;
 13 : STORPLACE DUP 4 PICK 2 * 1+
 14 CHARS ! OVER 4 PICK 2 *
 15 CHARS ! PLACEIT ; -->
```

Fig. 8.9. Screen 18.

This begins with definitions of the variables that will be needed in the game:

SCORE Obviously, this is where your score is to be held.

MOVEX This decides the amount by which the UFO can move each time in a left-right direction. Values in MOVEX are to be used in conjunction with MOVEIT, the word

described in Chapter Six. In this game, MOVEX is either −1 (move 1 space left) or 1 (move 1 space right).

MOVEY This decides the up-down motion of the UFO. It is either 1 or −1. Which of these gives up and which gives down depends on how your micro counts the rows of its screen. As used in this application, which is for the BBC Microcomputer and Electron, 1 gives downward motion and −1 gives upward motion.

YOUX and YOUY are the corresponding variables for your spacecraft.

Next we have to define an array to hold the co-ordinates of the UFO and your craft which we shall in future refer to as YOU. The array is to have 4 locations:

0	X-position of UFO
1	Y-position of UFO
2	X-position of YOU
3	Y-position of YOU

This is exactly the same scheme as in the array CHARS of Chapter Six. This array is called CHARS, too. The difference is that this CHARS needs only four locations. Before we can have an array, we define the word to define the array. This is the same word, ARRAY, as used in Chapter Six. Line 4 of Screen 18 (Fig. 8.9) has the definition of ARRAY. This is used on line 5 to define CHARS.

Next we will define the special characters of the UFO and YOU (Fig. 8.10). You can, of course, substitute designs of your own for these. Lines 6 to 10 are taken up with the definition of CHARDEF and using it to define the two characters. If you had to use your own version of CHARDEF in Chapter Six, to suit your micro or your FORTH, use it again here. Another such word is PLACEIT (lines 11–12). Key in your own definition, if it is different from the one

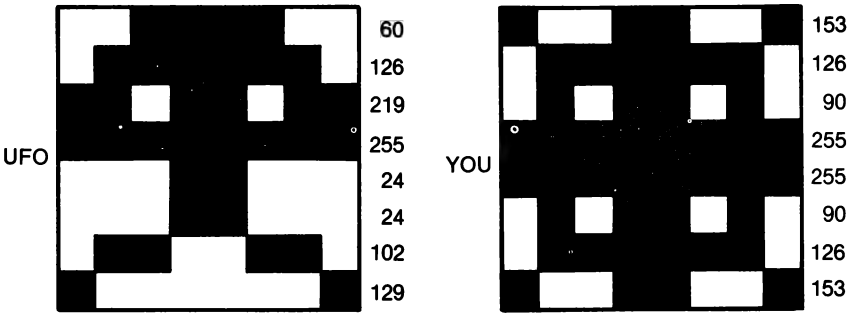


Fig. 8.10. Special characters for the UFO game.

shown. From this point on, except for the initialising word GO, we shall be using only words that are likely to be found in all versions of FORTH. The only changes needed will be those which allow for the number of rows and columns on your screen and the direction in which the rows are numbered.

Screen 18 ends with the definition of STORPLACE, exactly as in Chapter Six.

PART 2 (Screen 19) (Fig. 8.11)

```
SCR # 19       13 H
  0 ( UFO GAME - PART 2 )
  1 : BLANKIT -192 SWAP DUP 2 * CHARS @
  2 SWAP 2 * 1+ CHARS @ PLACEIT ;
  3 : MOVEIT 3 PICK 2 * 1+ CHARS @ +
  4 SWAP 3 PICK 2 * CHARS @  + SWAP
  5 STORPLACE ;
  6 : LEFT 0 CHARS @ 0= IF 1 MOVEX ! THEN ;
  7 : RIGHT 0 CHARS @ 38 = IF -1 MOVEX ! THEN ;
  8 : UP 1 CHARS @ 0= IF 1 MOVEY ! THEN ;
  9 : DOWN 1 CHARS @ 31 = IF -1 MOVEY ! THEN ;
 10 : BORDER LEFT RIGHT UP DOWN ;
 11 : GO 4 MODE 0 0 0 4 0 19
 12   0 0 0 13 1 19 0 0 0 0 0 0 0 0
 13   1 23 22 0 DO >VDU LOOP ;
 14
 15 -->
```

Fig. 8.11. Screen 19.

This begins with two more words from Chapter Six, BLANKIT and MOVEIT. Next come four words to alter the direction of motion of the UFO when it comes to the edges of the screen. In these we assume that the rows of the screen are numbered 0 to 31 from top to bottom, and the columns are numbered 0 to 39 from left to right. Actually we do not use column 39. Placing a character at the bottom of this causes a line-feed which upsets the display. LEFT changes the direction of motion when the UFO hits the left margin of the screen. '0 CHARS @' copies the X-position of UFO from CHARS and puts it at top-of-stack. '0=' tests to see if it is equal to zero, this being the left-most column. If your columns are numbered from 1, substitute '1 =' for '0='. This is followed by an IF...THEN routine. If the value is zero (UFO is at left of screen) the value 1 is stored in MOVEX. The effect of this will be to make UFO move toward the right when MOVEIT is next used.

RIGHT operates in a similar way, giving MOVEX the value −1 (move left) if UFO is in column 38. If your screen has a different

width, put the appropriate number in place of the 38. UP detects if UFO is at the top of screen, by first putting the Y-position (1 CHARS) on the stack. It tests to see if this is zero. Again, you may have to alter this value if your screen row numbering is different. If UFO is at top of screen, MOVEY is made 1, to make UFO move down when MOVEIT is next used. If your screen works the other way round, use −1 here. Similarly DOWN, checks if UFO is at bottom of screen, and alters MOVEY to make it move up. BORDER puts all four words together to test if UFO is at the borders of the screen and alters MOVEX or MOVEY (or possibly both) accordingly.

The final word on Screen 19 is GO, which initialises the screen ready for the game. This version is for the BBC Microcomputer and Electron. It does the following:

Sets screen Mode to 4.
Sets screen colour to blue (the equivalent of VDU 19,0,4,0,0,0 in BBC BASIC).
Sets foreground colour to flashing green/magenta (the equivalent of VDU 19,1,13,0,0,0).
Disables the cursor so that it does not spoil the display (VDU 23,1,0,0,0,0,0,0,0,0).

Some of these may not apply to your micro but, if they do, the FORTH handbook should explain how to perform equivalent operations. The Mode statement referred to above automatically clears the screen. Include this action in GO if you have not used MODE. Some FORTHs have a word CLS. An alternative is '12 EMIT' which works on most machines which recognise standard ASCII codes.

PART 3 (Screen 20) (Fig. 8.12)
PLAY sets up the array and variables ready to begin the game. CHARS is set to hold 0 and 0 for the UFOs X- and Y-positions, placing it at top left of screen. You may need to adapt these for your computer's screen. It is also set to hold 36 and 22 for YOU X- and Y-positions, placing YOU near the bottom right corner. The MOVEX and MOVEY are each given the value 1, to start UFO moving diagonally down the screen and to the right. You may need to make MOVEY −1 if the rows of your screen are numbered from the bottom upward. The score is set to zero.

We are now ready to put UFO into motion. The word MUFO (Move UFO) checks if UFO is at the edge of screen, using

```
SCR # 20       14 H
  0 ( UFO GAME PART 3 )
  1 : PLAY 0 0 CHARS ! 0 1 CHARS !
  2       36 2 CHARS ! 22 3 CHARS !
  3       1 MOVEX ! 1 MOVEY ! 0 SCORE ! ;
  4 : MUFO BORDER 0 MOVEX @ MOVEY @ MOVEIT ;
  5 : DELAY 100 0 DO LOOP ;
  6 : HALT 0 YOUX ! 0 YOUY ! ;
  7 : KEY/ 47 = IF 3 CHARS @ 31 =
  8                IF ELSE 1 YOUY !
  9                THEN
 10             ELSE THEN ;
 11 : KEY; DUP 59 = IF 3 CHARS @ 0=
 12                    IF ELSE -1 YOUY !
 13                    THEN DROP
 14                 ELSE KEY/ THEN ;
 15  -->
```

Fig. 8.12. Screen 20.

BORDER. In Chapter Six it was explained that MOVEIT requires the stack to have:

Character number	Amount of motion in X	Amount of motion in Y

The 0 is the character number of UFO. MOVEX @ and MOVEY @ place the other required values on the stack. Then MOVEIT displays UFO in its new position, and stores its new position in CHARS.

Line 5 is a delay loop, which you may or may not need, depending on how skilful you become at playing the game. It is possible to let the upper limit be a variable instead of 100, so that the speed of the game could be set by the player.

The next six words are concerned with moving YOU. The operative word is MYOU, in Screen 21 (Fig. 8.14), but we first define four words to be included in this. HALT is a word to set YOUX and YOUY to zero. It is used after each move by YOU so that YOU remains stationary until a key is pressed. The words KEY/, KEY;, KEYX and KEYZ; are all concerned with detecting key-presses and acting accordingly. KEYX and KEYZ are on Screen 21. The last character in each of their names indicates the key which the word tests. The words are linked together so that, as soon as a key press has been identified, the remaining keys are not tested. The sequence is as follows:

MYOU uses ?KEY to wait a short while for a key-press. The ASCII code code of the key is placed on the stack.

KEYZ tests to see if the key pressed is Z (code 90). If not, it calls KEYX. If the key is Z and YOU is not at the extreme left of the screen, YOUX is given the value −1, to move YOU to the right. The action continues without further examination of the key code. We will look into the action of KEYZ in more detail later.

KEYX tests to see if the key pressed is X (code 88). If not, it calls KEY;. If the key is X and YOU is not at the right of the screen, YOUX is given the value 1, to move YOU to the right. The action then continues.

KEY; tests to see if the key pressed is ; (semicolon). If not, it calls KEY/. If the key is ; and YOU is not at the top of screen, YOUY is given the value −1 to make YOU move up. The action then continues.

KEY/ tests to see if the key pressed is /. If not, the action continues, for all possible key presses have now been looked for. If the key is / and YOU is not at the bottom of screen, YOUY is given the value 1 to make YOU move down. The action then continues.

The operation of KEYZ is illustrated by the flow-chart of Fig. 8.13. It begins with the ASCII code of the key on the stack. DUP copies the code, so that one copy may be used in the test, and the original value left on the stack for use by one of the other 'KEY' words. Then '90 =' tests to see if the key was Z. We now have two nested IF...ELSE...THEN routines. The first IF detects if the key is Z. If so, it proceeds to find out if YOU is at the left of screen. '2 CHARS @ 0=' puts the X-position of YOU on the stack, then tests to see if it is equal to 0, the number of the left-most column. The inner IF then comes into play. If YOU is at the left, then no motion is possible. Nothing is done except to skip to the inner THEN. This is followed by DROP, to get rid of the unwanted ASCII code from top-of-stack.

If YOU is not at left of screen, motion is possible and the inner ELSE gives YOUX the value −1, to cause motion to the left. The action passes to the inner THEN and after this the unwanted ASCII code is DROPped.

If the condition at the outer IF is not met because the code is not 90, the outer ELSE comes into play. This sends the computer off to word KEYX, to find out if the X key was pressed. Whichever way

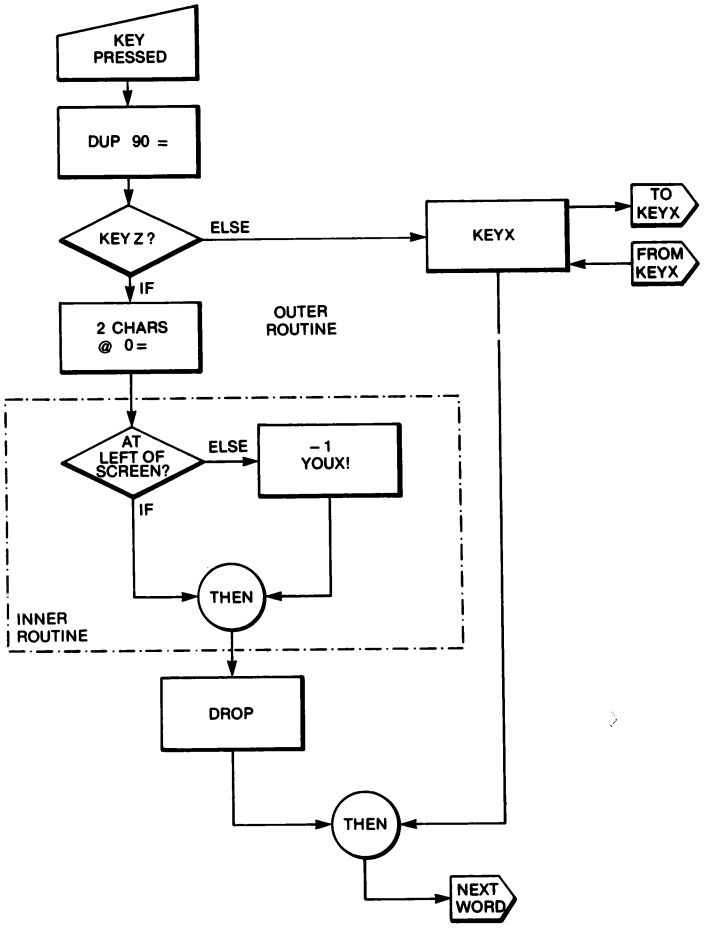


Fig. 8.13. Nested IF ... ELSE ... THEN routines in KEYZ. The inner routine is nested in the IF branch of the outer routine (compare with *Fig. 8.5).*

the action goes, it finishes at the outer THEN, and from there the computer passes to the next word in MYOU.

The structure of KEYX and KEY; are the same as that of KEYZ. KEY/ is a little simpler because, being the last word to be called, it does not need to duplicate the code, and there is no value to be DROPped at the end. Also, there is nothing in the outer ELSE section.

These words have illustrated several aspects of decision taking in FORTH.

PART 4 (Screen 21) (Fig. 8.14)
Having defined the KEY words, in reverse order so that each one can

```
SCR # 21       15 H
  0 ( UFO GAME - PART 4 )
  1 : KEYX DUP 88 = IF 2 CHARS @ 38 =
  2                    IF ELSE 1 YOUX !
  3                    THEN DROP
  4                  ELSE KEY; THEN ;
  5 : KEYZ DUP 90 = IF 2 CHARS @ 0=
  6                    IF ELSE -1 YOUX !
  7                    THEN DROP
  8                  ELSE KEYX THEN ;
  9 : MYOU 10 ?KEY KEYZ 1 BLANKIT
 10  1 YOUX @ YOUY @ MOVEIT HALT ;
 11 : SAME 0 CHARS @ 2 CHARS @ =
 12        1 CHARS @ 3 CHARS @ =
 13   0= IF DROP 0 ELSE 1 = THEN ;
 14 : HITUFO SAME IF SCORE @ 1+
 15  SCORE ! 7 EMIT THEN ; -->
```

Fig. 8.14. Screen 21.

call the earlier ones, we come to MYOU. This uses ?KEY to detect the key-press, the figure 10 allowing waiting period of 0.1 second. If your FORTH has INKEY, which does not wait, use this instead of '10 ?KEY'. The essential point is that the word used should not wait indefinitely for a key-press. This is followed by KEYZ, which calls all the other KEY words, if necessary. YOUX and YOUY will now contain values to produce the motion required. '1 BLANKIT' clears the previous image of YOU from the screen. Then the values required by MOVEIT are put on the stack:

1 the code for YOU character
YOUX
YOUY

MOVEIT displays YOU in its new position and stores this position in CHARS. HALT restores MOVEX and MOVEY to zero, so that YOU does not move until another key-press is detected.

One vital element of game programs is *coincidence detection*. We often want to know if two objects are in collision – if a bullet has hit its target, for instance. Here we want to know if UFO and YOU are at the same position on the screen. Coincidence is detected by SAME, which is another example of decision-taking. It is also a word that can be applied in many other games programs. The UFO and YOU are in collision if both have the same X-position (0 and 2 in CHARS) and both have the same Y-position (1 and 3 in CHARS. First, SAME puts the X-positions on the stack and tests to see if they

are equal. If they are equal, 'equals' leaves true (1) on the stack. If they are unequal, it leaves false (0). Next, SAME puts the Y-positions on the stack and tests these to see if they are equal, leaving a second flag on the stack. There are now four possible pairs of flags on the stack:

X-posns	Y-posns	Stack	
Different	Different	0	0
Same	Different	1	0
Different	Same	0	1
Same	Same	1	1

The top-of-stack is on the right. If YOU and UFO are in collision, there are two 1s on the stack, as in the bottom row of the table. The final section of SAME (line 13) tests top-of-stack to find out if it is zero using 'zero-equals'. If this is true then the stack is as shown in one of the *top* two lines of the table above. There is no collision. The 0 at top-of-stack has already been used by 'zero-equals' so all that is left to be done is to DROP the other value (0 or 1) and continue to THEN. If the result of 'zero-equals' is false, then the situation must be one of the *bottom* two lines of the table. If there is a collision, the flag remaining on the stack is 1. The ELSE action tests to see if this is 1 by using 1 =. If it is 1, then 'equals' leaves 1 on the stack. If it is 0, then equals leaves 0 on the stack. The result of SAME is to leave 1 on the stack, if there is a collision, otherwise it leaves 0. In short, it gives a true/false answer to the question 'Are they in the same position?'

Same is used by two words. HITUFO decides if YOU have hit the UFO. SAME returns a 1 if there is a hit, in which case the IF action is to increment SCORE and make the loudspeaker beep. If your FORTH has a BEEP command or any other special noise-making word, use this in place of '7 EMIT'. Perhaps you can invent other words to produce sound at this point.

PART 5 (Screen 22) (Fig. 8.15)
This begins with HITYOU which detects if UFO collides with YOU. It has the same general structure as HITUFO, except that your score is decremented, and there are two beeps. In practice, the two beeps run into one beep of double length.

FINISH is the word for ending the game. '12 >VDU' clears the screen. Your FORTH may have an alternative word for doing this. '5 0 DO CR LOOP' does five carriage returns. The purpose of this is to display the score a little way down the screen. Then comes the message announcing the score, after which SCORE is fetched to the

```
SCR # 22       16 H
  0 ( UFO GAME - PART 5 )
  1
  2 : HITYOU SAME IF SCORE @ 1-
  3   SCORE ! 7 EMIT 7 EMIT THEN ;
  4
  5
  6 : FINISH 12 >VDU 5 0 DO CR
  7   LOOP ." YOUR SCORE IS "
  8   SCORE @ . 20 0 DO CR LOOP ;
  9
 10
 11 : UFO GO PLAY 2000 0 DO 0 BLANKIT
 12   MUFO HITYOU DELAY MYOU HITUFO
 13   DELAY LOOP FINISH ;
 14
 15
```

Fig. 8.15. Screen 22.

stack and then displayed. Twenty more carriage returns follow, so that the 'OK' prompt is displayed at the bottom of the screen.

Now everything is ready for the final assembly of the UFO game. In true FORTH fashion, the complete routine for the game is contained in the one word UFO:

GO sets up the screen and colours.
PLAY initialises positions and other values.
Within the loop which repeats 2000 times we have:
'0 BLANKIT' to clear UFO.
MUFO to move UFO to its next position and display it.
DELAY to give you time to see UFO, and to slow the game down to a reasonably playing rate.
HITYOU to find out if UFO hit YOU.
MYOU to detect your key press and move YOU accordingly.
HITUFO to find out if YOU hit UFO.
DELAY to give you time to see YOU.

After the loop is over the game terminates with: FINISH. When all is ready, type UFO, press RETURN, and the chase commences!

Variations

There is endless scope for variations. Alter the characters or the colours. Use other ASCII codes so that YOU is controlled by the

keys of your choice. On the BBC Microcomputer or Electron, for example, the editing keys might be a better choice. Alter the speed of the game by varying the length of DELAY. The whole game could be put into a loop, allowing it to be replayed by pressing a key. At each replay the delay could be shortened. This would need a variable for length of delay, which was decremented each time the game was played. You could also keep a record of the maximum score. Some FORTHs have a word MAX which, given two numbers on the stack, removes both and replaces them with the greater of the two. Display this maximum each time along with the score of the current game.

There is also scope for experimenting with the overall structure of the game. You may find that UFO would benefit from rearrangement of the main words. Perhaps extra DELAYs would improve it. Maybe it would be better to use MYOU and HITUFO twice, to give the player more chances of moving.

One of the advantages of FORTH is that you can redefine some of the earlier words, perhaps adding extra features to them, without necessarily having to alter the words which call them. If you do alter the earlier words, remember that you must repeat the definitions of words which call them. If you do not do this these words will continue to use the old definitions of the earlier words. Having edited the screen and SAVEd it, reLOAD it to point the later words on to the newly-defined earlier words.

To summarise

In this chapter you have found out how to:

- Use flags to make the computer branch to one of several different routines.
- Write words to check that the user presses one of a specified set of keys.
- Round numbers to the nearest integer.
- Avoid errors by the user during input routines.
- Design games applications with graphics displays controlled from the keyboard.

You have used these FORTH words:

- IF ... THEN (flag ...) used in the form <flag> IF <action> THEN <continue>.

- IF ... ELSE ... THEN (flag ...) used in the form <flag> IF <action 1> ELSE <action 2> THEN <continue>.
- =>< equals, greater-than and less-than, the conditional operators for comparing two numbers (n1\n2 ... flag).
- 0= 0> 0< 'zero-equals', 'zero-greater', and 'zero-less', for comparing a number with zero (n ... flag).
- C! 'C-store' for storing a byte at a given address (byte\address ...)
- R: R; 'R-colon' and 'R-semicolon' for beginning and ending definitions of recursive words.
- NOT (flag ... inverted flag).

You have learned that:

- That the truth or falseness of a condition can be represented by a value on the stack, called a flag.
- The true flag is 1, though any non-zero value is recognised as true by the computer.
- The false flag is 0.
- A word or routine which calls itself is described as recursive.

Explore more

(1) Define a word which, given two values on the stack, displays the greater of the two.

(2) Define a word which, given two values on the stack, divides the greater value by the smaller one and displays the result.

(3) Extend the definition of YN? so that, if the user types a letter other than Y or N, a message such as "Please key Y or N" is displayed. The routine then waits for one of the correct keys to be pressed.

(4) Carry out some of the suggestions in the section called *Variations*.

Chapter Nine
Over and Over

No, the title of this chapter is not a reference to the stack manipulator, OVER, but introduces the theme of repeated *actions*. Often we want the computer to perform an action many times. Like many other languages, FORTH has looping structures which provide the repetition. Repetition is so important that we have already made frequent use of one such structure, the DO ... LOOP loop. Before going on to see what other kinds of loop are possible, here is a short summing up of the DO ... LOOP loop.

Figure 9.1 is a flow chart of the essential features of the loop. The

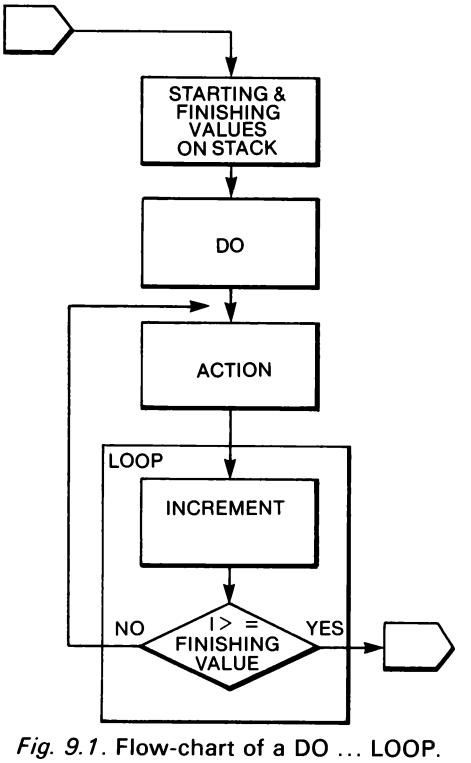


Fig. 9.1. Flow-chart of a DO ... LOOP.

loop uses a *loop index*, I, which is incremented each time the loop is repeated. Before the loop is begun, the finishing and starting values of this index are placed on the stack. The finishing value is at second-on-stack and the starting value is at top-of-stack. DO reads these two values. The action which is to be repeated usually consists of one or more words placed between DO and LOOP. Occasionally there may be no words between DO and LOOP, when we merely want to delay the computer, so as to slow down an application.

When the computer gets to LOOP, it increments I. Then it tests I to find out if it is equal to or greater than the finishing value. If it is not, the action is repeated. This continues, with I being incremented each time round the loop until, eventually, I is equal to or greater than the finishing value. There are no further repetitions of the action. Note particularly that there is no Ith repetition.

One important feature of loops is that they cannot be initiated immediately, by typing direct commands on the keyboard. They must always be part of a definition of a word. The word COUNTING demonstrates how the loop operates:

```
: COUNTING 6 0 DO I . CR
           LOOP ;
```

The action consists of putting I on the stack and then displaying it, followed by a carriage return (see Fig. 9.2).

```
COUNTING
0
1
2
3
4
5
OK
```

Fig. 9.2.

The value of I is incremented by 1 each time around the loop, beginning with the starting value (0), and ending with the finishing value (6). However, the loop ends as soon as LOOP detects that the finishing value has been reached, so there is no repetition when I equals 6. Only the numbers 0 to 5 are displayed.

LOOP increments I by 1. To increment I by other amounts, we use +LOOP. The amount by which it is to be incremented is to be placed on the stack immediately before using +LOOP:

```
: COUNTING 6 0 DO I . CR
             2 +LOOP ;
```

In this version of COUNTING, the finishing and starting values are the same as before, but I is to be incremented by 2 each time. The loop is repeated 3 times (see Fig. 9.3). The loop terminates when I becomes 6 and then equals the finishing value.

```
COUNTING
0
2
4
OK
```

Fig. 9.3.

A negative value may be used with +LOOP to decrement I each time:

```
: COUNTING 0 6 DO I . CR
            -1 LOOP ;
```

We have reversed the values, making I start at 6 and finish as 0:

```
COUNTING
6
OK
```

By mistake, the loop above used LOOP instead of +LOOP. The −1 was ignored by LOOP, which simply incremented the starting value (6) to 7. This was already greater than the finishing value, so there was no further repetition. Note that, whatever happens, whether the values placed on the stack make sense or not, a DO...LOOP loop or a DO...+LOOP loop always performs the action *at least once*. This is because the test for repetition comes *at the end* of the loop. The computer has to go through the action before it reaches this test for the first time.

Here is COUNTING properly redefined, using +LOOP:

```
: COUNTING 0 6 DO I . CR
            -1 +LOOP ;
```

Figure 9.4 shows its action. This example shows that the action of +LOOP differs from that of LOOP when I is being decremented. It repeats the action if I *equals* the finishing value, and stops only when I *exceeds* the finishing value. This gives an extra repetition that we do not get when I is being incremented.

```
COUNTING
6
5
4
3
2
1
0
OK
```

Fig. 9.4.

The word above incremented (or decremented) I by a fixed value. It is also possible to recalculate the amount of increment or decrement each time around the loop. This gives added flexibility. It is as if you could recalculate the STEP value of a FOR ... NEXT loop in BASIC which is not allowed. In this example the value is stored as a variable which, for reasons which will emerge in a moment, is called FIBO (see Fig. 9.5). FIBO is given an initial value

```
VARIABLE FIBO
OK
1 FIBO !
OK
```

Fig. 9.5.

1. The word which uses FIBO is FIBONACCHI, the name of a Tuscan mathematician of the twelfth century:

```
: FIBONACCHI 2000 0 DO FIBO @ I DUP
                        . FIBO !
                    +LOOP ;
```

The loop starts with I equal to 0 and is to run until it equals or exceeds 2000. The action of the loop is:

FIBO @	FIBO		
I	FIBO	I	
DUP	FIBO	I	I
	FIBO	I	
FIBO !	FIBO		
+LOOP			

The value of FIBO placed on the stack at the beginning of the loop is the value which I had during the previous loop. The *current* value of I is displayed and is stored away in FIBO ready for the next loop. At

the end of the loop, +LOOP adds the current value of I to the previous value of I (FIBO). The result of this repeated action is to give a FIBONACCHI series, in which each term is the sum of the two previous terms:

```
FIBONACCHI
0 1 1 2 3 5 8 13 21 34 55 89 144 233
377 610 987 1597 OK
```

The series stops at 1597 because the next value calculated for I takes it above its finishing value. Increasing the 2000 to some higher amount makes the calculation continue for more terms of the series.

Loops may be *nested* one inside the other, provided that the inner loop is completely enclosed by the outer one. Figure 9.6 shows an example. The outer loop is to start with its index equal to 0 and

```
: LOOPS 4 0 DO 9 6
              DO J . I . CR
              LOOP
           LOOP ;
```

Fig. 9.6

finishes with it equal to 4. The inner loop starts with its index equal to 6 and finishes with it equal to 9. Obviously two indexes are involved, one for each loop. With nested loops, the index I refers to the inner loop. Another index, J, refers to the loop surrounding the inner loop. The values of I and J may be used within the *inner* loop only, as in the example above. Figure 9.7 shows that for each stage of

```
LOOPS
0 6
0 7
0 8
1 6
1 7
1 8
2 6
2 7
2 8
3 6
3 7
3 8
OK
```

Fig. 9.7.

the outer loop (J equals 0, 1, 2 and 3) the inner loop runs through its own stages (I equals 6, 7 and 8).

I has more uses than just counting the number of repetitions. We can make use of it in repeated calculations, as in this word which calculates a table of squares:

```
: SQUARES 10 0 DO I DUP DUP . * . CR
                LOOP ;
```

I is duplicated twice, filling the top three positions of the stack. One copy of I is displayed directly. The other two are multiplied together and their product, the square of I, is then displayed as in Fig. 9.8. It may happen that a calculation has to proceed until its result reaches

```
SQUARES
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
OK
```

Fig. 9.8.

a given value. We might want only the squares which are less than 40, for example. This could be arranged by setting the finishing value of the loop to 7. It is easy to work out in advance that 6 squared is 36, while 7 squared is 49. The action is not executed for I=7, so squares of numbers from 0 to 6 are displayed. If the calculation is a complicated one, especially if it includes unpredictable values that are calculated by other actions of the application, we may not be able to say in advance what the finishing value of I should be. In this case we can use the word LEAVE to escape from the loop (see Fig. 9.9). I and its square are calculated as before. The value of I-squared is left on the stack to be tested by 'greater-than' as described in

```
: SQUARES 10 0 DO I DUP DUP . *
                   DUP . 40 > IF LEAVE
                               THEN CR
                LOOP ;
```

Fig. 9.9.

Chapter Eight. If I-squared is greater than 40, a true flag is left on the stack. The IF causes LEAVE to be executed and the computer leaves the loop (see Fig. 9.10).

```
SQUARES
0  0
1  1
2  4
3  9
4  16
5  25
6  36
7  49
OK
```

Fig. 9.10.

The DO ... LOOP and DO ... +LOOP loops are known as *definite loops*. The number of times they will repeat is decided before the loop begins by placing two values on the stack. Only LEAVE is able to end the loop before I reaches its prescribed finishing value. The other loops described in this chapter are known as *indefinite loops*.

BEGIN ... AGAIN

The most indefinite of all indefinite loops goes on for ever! Or at

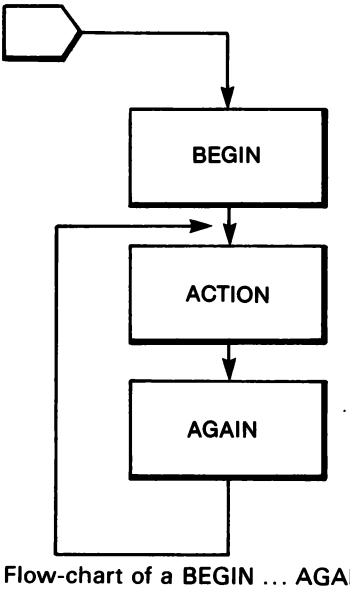


Fig. 9.11. Flow-chart of a BEGIN ... AGAIN loop.

least it would do if we did not have ESCAPE and BREAK keys and a power switch on the computer. The loop (Fig. 9.11) repeats the action for as long as the normal running of the computer continues. Here we use BEGIN ... AGAIN to display a message repeatedly:

```
: USER-FRIENDLY BEGIN ." HELLO"
                AGAIN ;
```

USER-FRIENDLY leaves no doubts about its friendliness (see Fig. 9.12). A BEGIN ... AGAIN loop is most often used for an

```
USER-FRIENDLY
HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLO
HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLO
HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLO
HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLO
Escape
OK
```

Fig. 9.12.

application which is to run repeatedly until the computer is switched off. For example, the UFO game might be enclosed in such a loop:

```
: SESSION BEGIN UFO KEY
          AGAIN ;
```

When the game is over, the computer waits for a key to be pressed. This gives the player time to read the score. The game is repeated when any key is pressed.

If your FORTH does not have BEGIN ... AGAIN, a practicable alternative is to use a DO ... NEXT loop with a suitable high finishing value. One can safely assume that most people would become bored with UFO (or exhausted!) and switch off before '20000 0 DO UFO KEY LOOP' runs to completion.

BEGIN ... UNTIL

This type of indefinite loop has many uses. It is equivalent to REPEAT ... UNTIL in BASIC. The word UNTIL is a testing word, similar to IF. A flag is put on the stack immediately before UNTIL is reached (Fig. 9.13(a)). If the flag is false, the action repeats, from BEGIN. The action of the loop is repeated UNTIL the flag is a true one. As with DO ... LOOP, the condition for ending the loop is

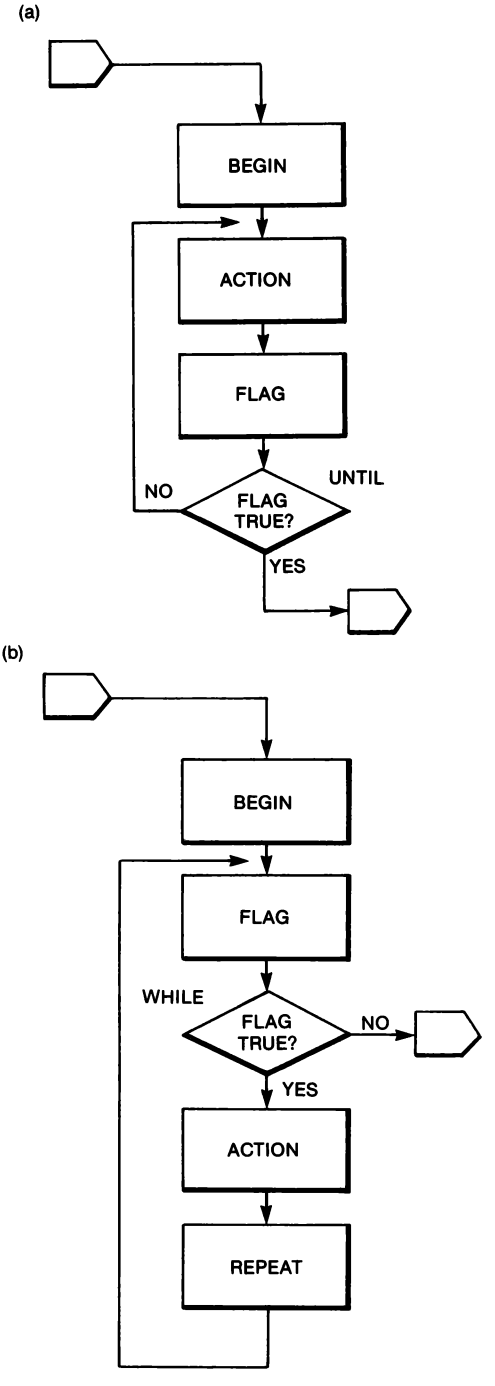


Fig. 9.13. Comparing (a) a BEGIN ... UNTIL loop with (b) a BEGIN ... WHILE ... REPEAT loop.

tested at the end of the loop. This means that the loop is executed at least once, even if the condition is true to begin with. Figure 9.13(b) illustrates the BEGIN...WHILE...REPEAT loop. This is discussed in Chapter Ten.

We will now design a reaction-time testing word, TEST, as an example of the use of BEGIN ... UNTIL. The variable TOTAL is to hold the total reaction time as it accumulates. RESET is used to reset TOTAL to zero before the test is to be performed. DELAY uses DO ... LOOP to give a delay of several seconds and, after that, displays the letter A on the screen (see Fig. 9.14).

```
VARIABLE TOTAL
OK

: RESET 0 TOTAL ! ;
OK

: DELAY 20000 0 DO LOOP 65 EMIT ;
```

Fig. 9.14.

The word to measure reaction time is REACTION:

```
: REACTION BEGIN TOTAL @ 1+ TOTAL !
                2 ?KEY 65 =
           UNTIL CR TOTAL @ . ;
```

The main part of it is a BEGIN ... UNTIL loop. The loop starts by putting TOTAL on the stack, adding 1 to it and storing the result as the new value of TOTAL. In short, TOTAL tells us how many times the loop has been repeated. The action of REACTION depends on ?KEY. As explained in Chapter Seven, ?KEY makes the computer wait for a definite length of time for a key to be pressed. If no key is pressed during that time the computer continues. The value 2 is put on the stack before ?KEY is used, making the computer wait for one fiftieth (two hundredths) of a second each time around the loop. The loop repeats indefinitely, until the A key is pressed. Each time round the loop, test the value left by ?KEY. This is done by placing 65 (the ASCII code for A) on the stack and then using 'equals'. If a true flag is left on the stack by 'equals', the loop is not repeated. The action of REACTION, so far, is to wait again and again for a key to be pressed until the A key is pressed. Then the computer leaves the loop and, after a carriage return, displays the latest value of TOTAL. This tells us how many fiftieths of a second have elapsed since the loop was started.

The words described above may now be put together to make up the reaction test:

```
: TEST RESET DELAY REACTION ;
```

The first thing is to reset TOTAL. Then DELAY makes the computer pause, before displaying A. The user will have been watching the screen and, as soon as A appears there, is to press the A key. REACTION will have started its looping meanwhile. During the time that the user requires for noticing the A on the screen and then finding and pressing the A key, the loop will have been repeated several times. The number of loops is displayed as soon as A is pressed.

```
TEST
A
16 OK
```

The example above shows the sequence of events. TEST is keyed in and RETURN is pressed. Nothing happened for a few seconds, until the A appeared. The A key was pressed immediately after that, but not before the loop had repeated 16 times. The user's reaction time is 16/50 second, or 0.32 second. You could add routines to TEST to display the final result in seconds or hundredths of a second. Test your own reaction times – it should be easy to beat the result shown above.

YN? revisited

In Chapter Eight we devised a word YN? which waits until the user has typed either Y or N. It made use of recursion to repeat the input sequence until one or the other of these keys was pressed. There were certain unsatisfactory aspects to this approach and it is now clear that a BEGIN ... UNTIL loop could provide a better solution to the problem. We need to repeat the keying in sequence *until* either the Y or the N has been pressed. The new definition of YN? given in Fig. 9.15 has a snag, as will be explained later, but it makes a starting point for exploring the possibilities.

The loop begins with KEY, which waits indefinitely for a key to be pressed and then puts its ASCII code on the stack. We are going to want to test the value left by KEY, to see if it corresponds to Y (ASCII code = 89), to test it again to see if it corresponds to N (ASCII code = 78) and finally to display the result as Y or N on the

```
: YN? BEGIN KEY DUP DUP 89 =
              SWAP 78 = = NOT
      UNTIL EMIT CR ;
OK


YN?
Y
OK


YN?
N
OK
```

Fig. 9.15.

screen. Or we might use the value (78 or 89) left on the stack for some other purpose. In all we need to have three copies of the ASCII code, and this is provided by 'DUP DUP'. The sequence of finding out if the Y or the N has been pressed is as follows (X is the value left by KEY):

The stack holds		X	X	X	
Test 1:	89	X	X	X	89
	=	X	X	Flag -1	
	SWAP	X	Flag -1	X	
Test 2:	78	X	Flag -1	X	78
	=	X	Flag -1	Flag -2	

At this stage the two flags may be:

	Flag -1	Flag -2
If Y pressed	1	0
If N pressed	0	1
If any other key pressed	0	0

If a key other than Y or N is pressed we want the loop to repeat. Note that it is not possible for *both* flags to be 1. The next 'equals' tests if the two flags are equal to each other, which in this case means that they are both equal to zero. If this is true, we want the loop to repeat. But UNTIL only repeats a loop if it finds a false flag at top-of-stack! The solution to this is to invert the flag, using NOT, as explained in Chapter Eight.

If neither Y nor N is pressed the false flag causes UNTIL to repeat

the loop. If Y or N is pressed, so that the flag is true, the loop ends. The value X left at top-of-stack is then displayed, using EMIT.

This word works as expected but is faulty in one important respect. Whenever a key other than Y or N is pressed its value (X) is left on the stack when the loop repeats. A series of values is gradually accumulated on the stack during the running of an application. It is a general rule that values must not be allowed to accumulate in this way, otherwise a 'Stack Full' error may be caused. Words should leave on the stack only such values as may be required by succeeding words. DROP seems a likely candidate for getting rid of the unwanted X, but the problem is where to use it. It must be used inside the loop, yet we cannot use it immediately before UNTIL. We do not know if X is to be dropped or not until UNTIL has tested the flag! The solution is to drop it at the beginning of the loop, immediately after BEGIN. Each time the loop repeats, the value X left on the stack by the previous loop is DROPped, before the next key-press is accepted. This new version of YN? begins by placing a dummy value (0, but it could be anything) on the stack ready for DROPping the first time DROP is encountered:

```
: YN? 0 BEGIN DROP KEY DUP DUP 89 =
              SWAP 78 = = NOT
        UNTIL EMIT CR ;
```

On the second and subsequent times round the loop the computer returns to BEGIN, not to the 0, so it is the value X which is dropped. This word leaves nothing behind on the stack.

Which number?

A similar technique can be used for accepting a numeric input within a given range. For example, the user might have been presented with a menu of choices numbered from 1 to 4. The word NO? is to accept any number within that range, but not outside it (see Fig. 9.16). The

```
: NO? 0 BEGIN DROP KEY DUP DUP 48 >
              SWAP 53 < =
        UNTIL EMIT CR ;
OK

NO?
3
OK
```

Fig. 9.16.

ASCII codes of 1, 2, 3 and 4 are 49, 50, 51 and 52 respectively. This word finds out if the code left at top-of-stack is greater than 48 and less than 53. As with YN?, two flags are left on the stack:

	Flag -1	Flag -2
key 0 or less	0	1
key 1	1	1
key 2	1	1
key 3	1	1
key 4	1	1
key 5 or more	1	0

'Key 0 or less' means pressing 0 or any punctuation key which has an ASCII code of 46 or less. 'Key 5 or more' means pressing key 5, or any greater number, or any letter or punctuation key with a higher ASCII code. It is not possible for both flags to be 0.

The loop is to be left only if both flags are 1. As in YN?, we use 'equals' to test if the two flags are the same. But here we want to end the loop if they are the same, so the NOT is not required. If they are the same, the loop ends and the number is displayed. If they are not the same, the loop repeats, dropping the value of the unaccepted key-press before waiting for the next input.

BEGIN ... UNTIL loops can be nested with each other or with other kinds of loop provided that the usual rules for nesting one loop completely inside the other are obeyed. Figure 9.17 shows an

```
: NO? BEGIN DROP KEY DUP DUP 47 >
            SWAP 58 < =
      UNTIL ;
OK
: NOS? 5 0 DO NO? DUP EMIT CR
             48 - I VALUES !
         LOOP ;
OK
```

Fig. 9.17.

example, NOS? which uses NO? to collect 5 numbers (each in the range 0 to 9) and place them in an array, VALUES. VALUES is defined, using ARRAY as explained in Chapter Six, to hold 5 values. The sequence of words within the BEGIN ... UNTIL loop of NO? is the same as in the previous version, except for the '47' and '58' which allow any number in the range 0 to 9 to be keyed in. Letters and symbols are not accepted. NOS? uses NO? in a DO...LOOP, so in effect we have a BEGIN...UNTIL nesting inside a DO...LOOP. After NO?, the value on the stack is duplicated so that it can be

```
NOS?
3
5
2
8
1
OK
2 VALUES @ .
2 OK
4 VALUES @ .
1 OK
```

Fig.9.18.

displayed. It is then reduced by 48, since the ASCII code of a number is 48 greater than the value of the numeral, and is then stored in VALUES. I is used to decide which location of VALUES it is stored in.

Repeat for ever

Some FORTHs lack BEGIN ... AGAIN, but this deficiency can easily be remedied by adapting the BEGIN ... UNTIL loop. The word immediately preceding UNTIL is 0:

BEGIN <action> 0 UNTIL

This places a false flag on the stack *every* time round the loop. As a result, UNTIL never finds true and the loop never ends.

To summarise

In this chapter you have found out how to:

- Use DO ... LOOP and the associated indexes I and J to repeat an action a definite number of times.
- Use BEGIN ... AGAIN to repeat an action indefinitely.
- Use BEGIN ... UNTIL to repeat an action until a given condition is true.
- To check that the user has pressed one of a given set of keys.

You have used these FORTH words:

- DO comes before the action words of a definite loop (n1\n2 ...).

- LOOP ends the action sequence of a definite loop; action repeats from DO (...).
- I the loop index (... I).
- J the loop index of the outer of two nested loops (... J).
- +LOOP used instead of LOOP when I or J are to be stepped down (decremented) or to be stepped up (incremented) by more than 1 (step ...).
- BEGIN comes before the action words of an indefinite loop (...).
- AGAIN ends the action section of an indefinite loop; the loop repeats from BEGIN (...).
- UNTIL ends the action section of an indefinite loop when a given condition is true (flag ...).

Explore more

(1) Write a word to display a table of numbers and their cubes, for all the number from 1 up to the largest number with a cube less than 30000.

(2) Write a word to check that a number of one or more digits, typed in by the user, lies within a given range, for example (1–99).

(3) Devise a simple game along the following lines, and write the words needed for it. An aeroplane flies repeatedly across the screen from left to right. Each time it crosses the screen the player's score is increased by 1. The player has to shoot the aeroplane down by pressing the space-bar at the exact moment the aeroplane reaches the centre of the screen. If the key is pressed at the right time, the plane explodes, the game ends and the player's score is displayed. If the player presses the key too early or too late to hit the aeroplane, the player's score is increased by 1. The object of the game is to obtain a very small score.

Chapter Ten
Sorting Numbers

It is often useful to be able to sort a set of values into numerical order. Sorting is an operation in which the same action has to be repeated over and over again. As one might expect, repeating loops play a major role in sorting routines.

The routines developed in the remainder of this chapter are to be generally applicable. All values to be sorted will be held in an array, called VALUES. Screen 23 (Fig. 10.1) shows the definition of the defining word ARRAY, which is exactly the same as used in earlier chapters.

```
SCR # 23        17 H
  0 ( SORTING - PART 1 )
  1 : ARRAY CREATE 2 * ALLOT
  2           DOES> SWAP 2 * + ;
  3 5 ARRAY VALUES
  4
  5 VARIABLE NEXT
  6 : SET 37 0 25 1 48 2 21 3 30 4 5 0
  7         DO VALUES ! LOOP ;
  8 : SHOWVAL 5 0 DO I VALUES @ . LOOP ;
  9
 10 : COMP DUP VALUES @ ROT DUP
 11          VALUES @ ROT OVER OVER
 12      > IF ROT VALUES ! SWAP VALUES !
 13        ELSE 2DROP 2DROP THEN ;
 14
 15 -->
```

Fig. 10.1. Screen 23.

Next we define the array VALUES, to hold 5 numbers. There is no limit on how many numbers it can hold, except that set by the amount of free memory available. Five values are enough for showing how the sorting routines work.

The variable NEXT is defined next. Its purpose will be described later.

SET is to be used to place a set of unsorted values into VALUES quickly, so that we can check the action of the application as we develop it. It is not needed when the application is finished. SHOWVAL too is not needed for sorting. but is used to display the contents of VALUES. This is handy for following the sorting stage-by-stage.

The essential feature of most sorting procedures is to take two values, compare them and, if necessary, swap them round. The word COMP does just this. It is designed to take any two values from VALUES and then to place the lesser value in the location of lower number and the greater value in the other location. For example, it might take the values from location 4 and location 6. These might be

```
Location 4    Value held  =  321
Location 6    Value held  =  123
```

In this case the values would be swapped so that the lesser value (123) goes into the location with smaller number (location 4). If the values held were as follows:

```
Location 4    Value held  =  111
Location 6    Value held  =  555
```

Then there would be no need to swap, as location 4 already holds the smaller value. COMP is designed for routines which sort numbers in ascending order. You could easily adapt it to sort the other way round.

COMP requires two values on the stack; the numbers of the locations to be compared, with lower number at second-on-stack and higher number at top-of-stack. Let us call these n1 and n2 respectively. The values which are held in these locations will be referred to as (n1) and (n2). Here is how COMP works (top-of-stack to right, as usual):

```
COMP  expects      n1   n2
      DUP          n1   n2   n2
      VALUES @     n1   n2  (n2)
      ROT          n2  (n2)  n1
      DUP          n2  (n2)  n1   n1
      VALUES @     n2  (n2)  n1  (n1)
      ROT          n2   n1  (n1) (n2)
      OVER OVER    n2   n1  (n1) (n2) (n1) (n2)
```

```
        >           n2   n1   (n1)  (n2)  flag
IF                  n2   n1   (n1)  (n2)
        ROT         n2   (n1) (n2)  n1
        VALUES !    n2   (n1)
        SWAP        (n1) n2
        VALUES !
```

The above shows what happens if the values are in the wrong order and so need to be swapped. If 'greater-than' (>) leaves a true flag, the IF branch of the routine comes into operation. If the flag is false, there is no swapping.

ELSE finds 4 values on the stack. These are got rid of by using 2DROP twice. 2DROP is really intended for DROPping double-precision values from the stack (see Chapter Eleven), but we can use it here for DROPping two single-precision values at once.

Exchange sort

The first type of sort that we shall look at is the exchange sort. The way this operates is shown in Fig. 10.2. The rows of the diagram show the contents of VALUES, each time COMP has just exchanged two values. Ignore the words at the right-hand of the figure for the present.

The principle of this sort is as follows:

(1) Take the value in the first location and compare it with the values in the other locations in turn, starting with the value in the second location and proceeding to the end of the array. If the value in the first location is greater than the one it is compared with, exchange them. The result of this is to place the smallest value of all in the first location. Figure 10.2 shows two such exchanges, resulting in the smallest value (21) being placed in location 0.

(2) Repeat the above action but this time compare the value in the second location with all others in succeeding locations. Exchange when necessary. This puts the second smallest value (25) in the second location (1).

(3) Continue in this way comparing the third and fourth (last-but-one) locations with succeeding locations. Eventually the numbers will be in ascending numerical order.

Figure 10.2 shows the stages of sorting a particular sequence of

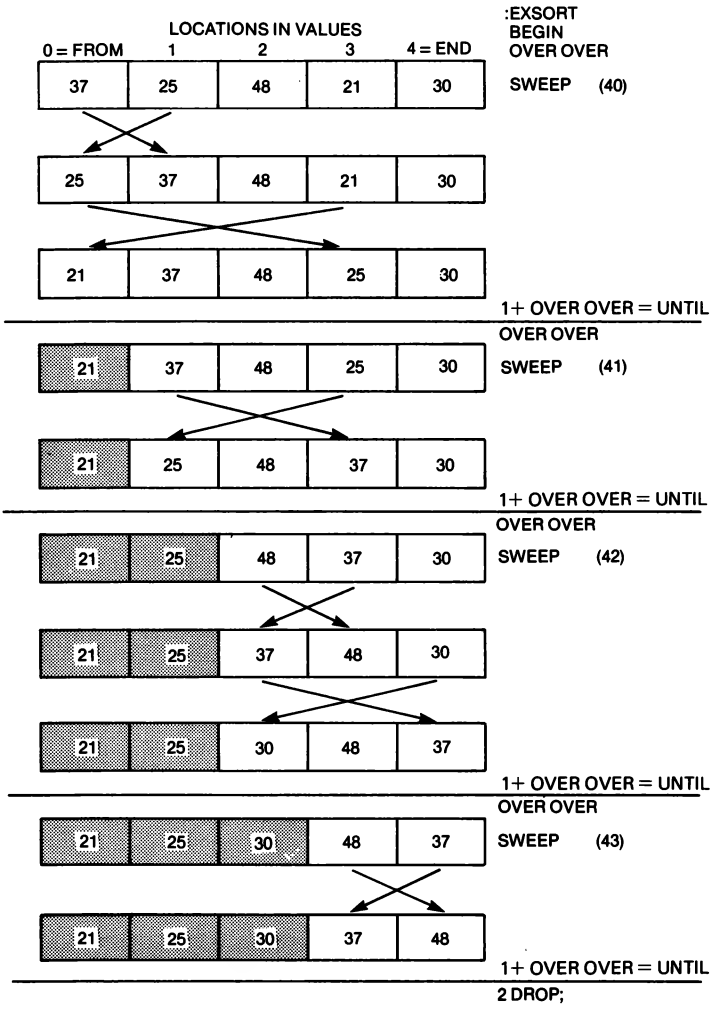


Fig. 10.2. Exchange Sort: (left) values in array VALUES after each swap. (Right) stages in the action of EXSORT. The numbers in brackets after SWEEP indicate the range of locations over which it sweeps. Shaded locations hold values sorted into their final places.

values. Let us put these numbers into VALUES as shown in Fig. 10.3. SHOWVAL proves that they are really there!

```
SET
OK
SHOWVAL
37 25 48 21 30 OK
```

Fig. 10.3.

Now we need a word which will perform the operation described in (1) above, beginning at any given location. The word for this purpose is SWEEP, which sweeps along the array, comparing and exchanging, when necessary, as it goes. This is listed in Screen 24 (Fig 10.4). The point of interest about SWEEP is that it uses a kind

```
SCR # 24        18 H
  0 ( SORTING - PART 2 )
  1 : SWEEP DUP 1+ NEXT !
  2   BEGIN OVER 1+ NEXT @ >
  3   WHILE DUP NEXT @ COMP NEXT @
  4         1+ NEXT ! REPEAT 2DROP ;
  5 : EXSORT BEGIN OVER OVER
  6   SWEEP 1+ OVER OVER = UNTIL 2DROP ;
  7
  8
  9
 10
 11
 12
 13
 14
 15  -->
```

Fig. 10.4. Screen 24.

of indefinite loop which we have not yet described. This is the BEGIN ... WHILE ... REPEAT loop. The action of this loop is shown in Fig. 9.13(b), where it can be contrasted with that of the BEGIN...UNTIL loop. The main difference is that whereas the BEGIN ... UNTIL loop tests a flag at the end of the loop, the BEGIN...WHILE...REPEAT tests it at the beginning.

The second difference is that the BEGIN...UNTIL loop repeats as long as the flag is false, and stops repeating when the flag is true. The BEGIN...WHILE...REPEAT loop repeats as long as the flag is true and stops repeating when it is false.

We have already said that a BEGIN...UNTIL loop will always perform its action at least once. A BEGIN...WHILE...REPEAT loop need never perform its action at all. If the flag is false when the loop begins, WHILE sends the computer straight out of the loop and the action is by-passed. SWEEP requires two variables on the stack. The number of the location at which the sweep is to begin will be referred to as f (for from). The number of the last location to be swept is called e (for end). SWEEP begins by placing a value in NEXT:

```
SWEEP  expects      e   f
         DUP        e   f   f
         1+         e   f   f+1
         NEXT !     e   f
```

To begin with, f equals 0 (first location), so NEXT now holds 1. This means that the first comparison is to be made between the values in the first and second locations.

The part of SWEEP between BEGIN and WHILE sets up the flag ready for testing by WHILE:

```
BEGIN       e   f
OVER        e   f   e
1+          e   f   e+1
NEXT @      e   f   e+1  (next)
>           e   f   flag
```

If the value of NEXT (listed above as (next)) is less than e+1, the sweep is to continue. The flag is true and WHILE portion of SWEEP will be executed. Note that e and f have been retained on the stack throughout all the above stages, ready for use in the loop:

```
WHILE       e   f
DUP         e   f   f
NEXT @      e   f   f      (next)
COMP        e   f
NEXT @      e   f   (next)
1+          e   f   (next)+1
NEXT !      e   f
REPEAT      e   f
```

The loop tells COMP (p.118) to compare the number in the first location with the number in the next location. Then NEXT is incremented. Note that e and f are left on the stack ready for BEGIN. In constructing a loop it is essential that the values required for setting the flag in the BEGIN... WHILE section are left on the stack just before the REPEAT.

During the loop, NEXT is incremented so that, as the sweep proceeds, the contents of location 0 are compared in turn with the contents of location 1, then with location 2, and so on to the end of the array.

SWEEP ends with 2DROP to get rid of e and f when the loop no longer repeats.

Now to see SWEEP in action. As explained earlier, the first sweep

is to take the value in location 0 and compare it with all others in successive locations. We need 0 (= f) and 4 (= e) on the stack (See Fig. 10.5). SHOWVAL shows that SWEEP has done its job. Location 0 now holds the smallest value, 21. Compare Fig. 10.5 with the third row of Fig. 10.2.

```
4 0 SWEEP
OK
SHOWVAL
21 37 48 25 30 OK
```

Fig. 10.5.

The next step it to take location 1 as the 'from' location (f = 1). The end is still 4. The second SWEEP is shown in Fig. 10.6. Only two more sweeps are needed to complete the sort (See Fig. 10.7).

```
4 1 SWEEP
OK
SHOWVAL
21 25 48 37 30 OK
```

Fig. 10.6.

```
4 2 SWEEP
OK
SHOWVAL
21 25 30 48 37 OK


4 3 SWEEP
OK
SHOWVAL
21 25 30 37 48 OK
```

Fig. 10.7.

Instead of using SWEEP by typing it in repeatedly, we have a REPEAT...UNTIL loop. This loop is the main part of the word EXSORT (short for Exchange Sort). The values of f and e are expected on the stack and these are copied by 'OVER OVER'. SWEEP uses the copies of f and e, the original f and e being left on the stack. When the sweep is finished, f is incremented by 1, so that the sweep begins one location further along the array each time it is repeated. This is just what we did above, when we used SWEEP repeatedly from the keyboard.

After using SWEEP, EXSORT adds 1 to the value of f (at top-of-stack) ready to repeat the sweep with a new starting location. However, if f now equals e, no further sweeps are required. So e and the new f are copied by OVER OVER and tested with 'equals'. If they are equal, the flag left on the stack is 1 and UNTIL does not cause a repeat. The remaining values of e and f are disposed of by 2DROP. The stages of action of EXSORT are indicated at the right-hand side of Fig. 10.2. Figure 10.8 shows EXSORT in action. The

```
SET
OK
4 0 EXSORT
OK
SHOWVAL
21 25 30 37 48 OK
```

Fig. 10.8.

time taken by EXSORT varies, depending on how many values are already in numerical order. When tested on an array of 101 values, already in order from 0 to 100, it takes just under 1 second (on the BBC Microcomputer). On an array of 101 numbers arranged in the reverse order (100 to 0), the time was 20 seconds. An equivalent routine in BASIC took 40 seconds.

Bubble sort

Figure 10.9 illustrates another commonly used sorting routine. In this one, we move (or sweep) along the array, comparing each value with the one next door to it and swapping them if necessary.

This process is repeated along *the whole length* of the array as many times as is necessary to place the values in the correct order. We need a flag variable, SWAPPED, which is set to 1 before each sweep. If two values are swapped, this variable is set to zero. At the end of each sweep, the variable is tested. If it is 1, the whole array has been swept with no swapping, indicating that sorting is complete. SWAPPED is defined in Screen 25 (Fig. 10.10).

Since SWAPPED has to be changed whenever an exchange occurs, we need a new version of COMP. This version, COMPS, is shown in Screen 25 (Fig. 10.10). It is exactly the same as COMP except for the inclusion of the extra words '0 SWAPPED !' in the IF branch of the routine.

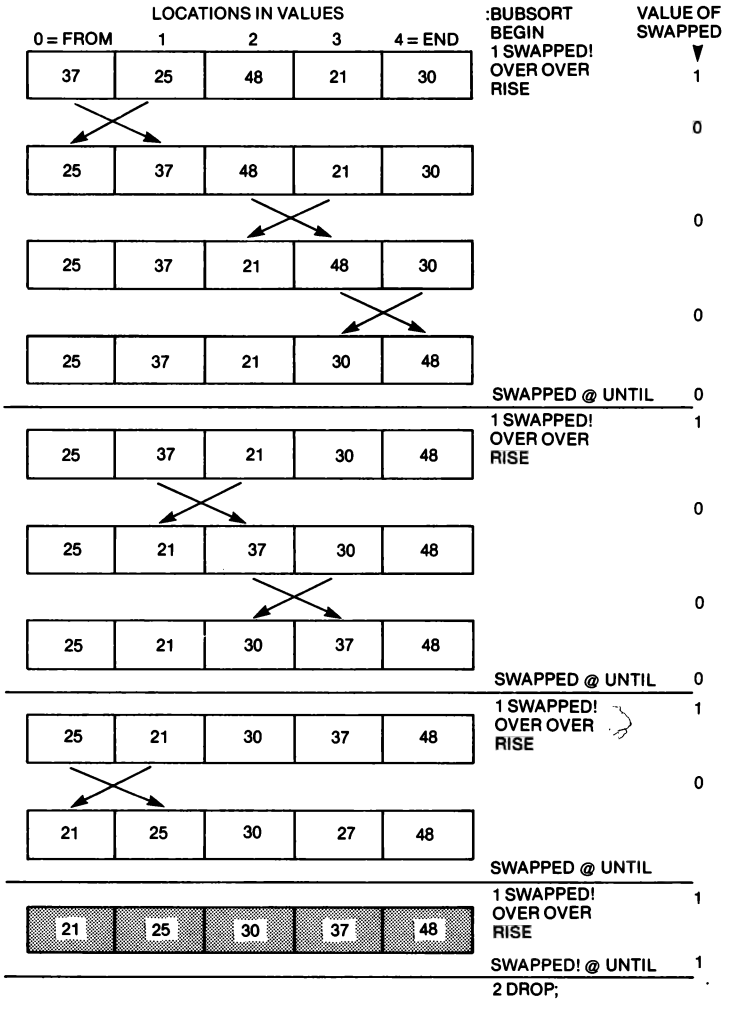


Fig. 10.9. Bubble Sort (left) values in array VALUES after each swap. (Right) stages in the action of BUBSORT. Shaded locations hold values sorted into their final places.

The Bubble Sort derives its name from the way in which the smaller values tend to rise toward the beginning of the array, like bubbles in a liquid. This is why we have called the sweeping word RISE. It requires the values of e and f on the stack. It is left to the reader to work out how it operates, by setting out the stack action as for the Exchange Sort words described above. Use SET to place values in the array as before and use RISE repeatedly, until SHOWVAL indicates that the sorting is complete. Compare the results at each stage with the rows of Fig. 10.9.

```
SCR # 25       19 H
  0 ( SORTING - PART 3 )
  1 VARIABLE SWAPPED
  2 : COMPS DUP VALUES @ ROT DUP
  3         VALUES @ ROT OVER OVER
  4     > IF ROT VALUES ! SWAP VALUES !
  5           0 SWAPPED !
  6       ELSE 2DROP 2DROP THEN ;
  7 : RISE BEGIN DUP DUP 1+ COMPS
  8     1+ OVER OVER = UNTIL DROP DROP ;
  9 : BUBSORT BEGIN 1 SWAPPED !
 10   OVER OVER RISE SWAPPED @
 11     UNTIL 2DROP ;
 12
 13
 14
 15  -->
```

Fig. 10.10. Screen 25.

BUBSORT provides the complete bubble sort routine. It begins by setting SWAPPED to 1. After RISE, it tests to see if it has changed. If there has been an exchange, SWAPPED is 0 (false) so the loop repeats. If all values are completely sorted and there has been no exchange, SWAPPED is 1 (true) so there is no repeat. 2DROP gets rid of the values of e and f remaining on the stack.

Both of these routines can be used with any pair of values for f and e. This makes it possible to perform sorting of several sets of values within the same application (See Fig. 10.11).

In the example below, we first sorted the values in locations 2 to 4.

```
SET
OK
SHOWVAL
37 25 48 21 30 OK
4 2 EXSORT
OK
SHOWVAL
37 25 21 30 48 OK
1 0 EXSORT
OK
SHOWVAL
25 37 21 30 48 OK
```

Fig. 10.11.

The values in location 0 and 1 are unaffected. Then we sorted the values in 0 and 1, independently of those in 2 to 4. The advantage of this is that you can set up a large array to hold all the sets of values that are to be sorted in a given application. Each set occupies its own range of locations and can be sorted independently. This makes it unnecesary to define separate words for sorting each of several differently named arrays.

Quicksort

Exchange Sort and Bubble Sort are useful when the number of items to be sorted is small. Their disadvantage is that as the number of items increases the number of comparisons to be made increases alarmingly. Doubling the number of values to be sorted increases the sorting time fourfold. Bubble Sort can be quick if the values are in more-or-less the right order to start with, but takes much longer than the Exchange Sort if the numbers are in reverse order. It has the disadvantage that exchanges always occur between adjacent locations so that a value can never move more than one place at a time along the array. A low number which begins at the top end of the array is going to take a long time to reach its destination.

Quicksort has a different approach to sorting, as illustrated in Fig. 10.12. Widely separated numbers can be exchanged, so movement is rapid. The first step is to choose a value called the *comparand.* For convenience, this may be taken as value in the 'middle location' of the array. If the array has an even number of locations, it is the last location in the first half of the array. In Fig. 10.12 the comparand (48) has a circle drawn round it. There are two variables, called the left pointer (LP) and the right pointer (RP). To start with, these point at the ends of the array. Thus LP = 0 and RP = 4 in this example. We begin by incrementing the value of LP until the value in the location it points to is equal to or greater than the comparand. In this example, we move LP to location 2, which contains the comparand itself since the values in locations 0 and 1 are both less than the comparand. Next we decrement RP until the value in the location it points to is equal to or less than the comparand. In this example, it already points to the value 30, which is less than the comparand, so no change is necessary. Later we shall see that a BEGIN...WHILE...REPEAT loop is used to move the pointers. The example of RP given above is a case in which no change is

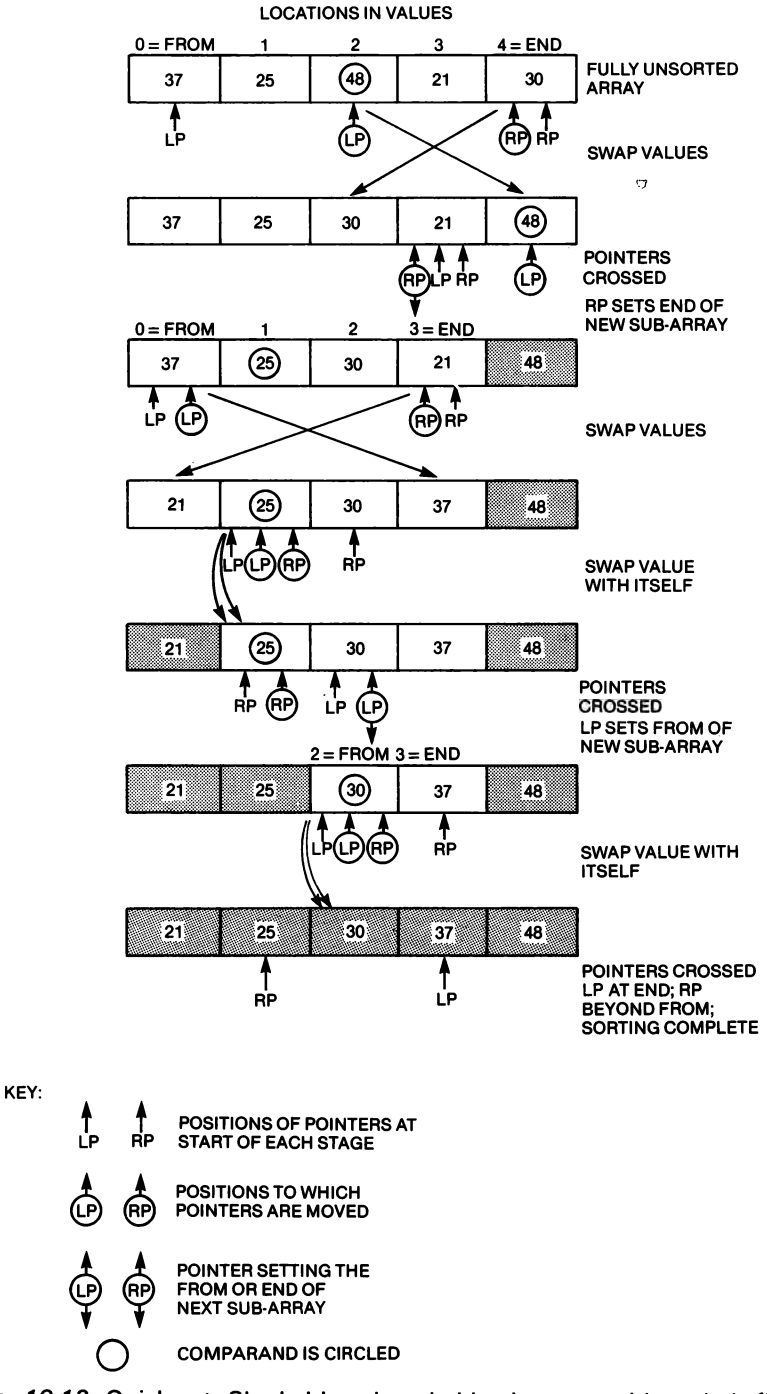


Fig. 10.12. Quicksort. Shaded locations hold values sorted into their final location.

required from the beginning, so a BEGIN...WHILE...REPEAT loop is just what is needed.

Next we look at positions of the pointers. If LP is still to the left of RP (LP<RP), we swap the values in these two locations of the array. This brings us to the second line of Fig. 10.12. At the same time we increment LP by 1, and decrement RP by 1.

The steps above are now repeated. LP moves to point to the 48, now at location 4. RP stays where it is, since 21 is already less than 48. When we look at the positions of the pointers we now find that they have crossed; LP is to the right of RP (LP>RP). This fact brings this part of the routine to an end.

The routine above covered the whole array. From now on the same routine is used, but is applied to part of the array. Which part is involved depends on the positioning of the pointers. If RP is not at or to the left of the left end of the array, the region extends from the left end to the location pointed at by RP. This is the case here, so the array is now considered to extend from 0 to 3. This can be thought of as a subarray which is about to be sorted, using the same procedure as described above. In short, the routine is a recursive one which calls upon itself to sort the subarrays which it creates.

If RP is not positioned as described above, we look at LP. If this is not at or to the right of the right end of the array (or subarray), the new range of the array extends from the position of LP to the end of the array. If the conditions above are not met by either pointer, the sorting is complete.

The above may sound rather complicated, which it is, but this method of sorting is extremely efficient and fast. You may have noticed in Fig. 10.12 that the array has been put in the correct order by the fourth line. The routine goes through two extra stages of swapping the comparand with itself before finally ending. This may seem wasteful, but with a larger array, the amount of time spent doing this is much less in proportion. The larger the number of values to be sorted, the more efficient Quicksort becomes.

The words for Quicksort are shown in Screens 26 and 27 (Figs 10.13 and 10.14).

The variables END and FROM are used to hold the numbers of the locations at the ends of the array (e and f in the previous description). As with the other sorts, we can use Quicksort on any chosen section of the array VALUES. The values held in FROM and END change as the array is divided into subarrays, and then refer to the subarray currently being sorted. Variables LP and RP are the left and right pointers respectively.

```
SCR # 26       1A H
  0 ( SORTING - PART 4)
  1 VARIABLE END  VARIABLE FROM
  2 VARIABLE LP  VARIABLE RP
  3 : LEFT BEGIN DUP LP @ VALUES @ >
  4   WHILE LP @ 1+ LP ! REPEAT DROP ;
  5 : RIGHT BEGIN DUP RP @ VALUES @ <
  6   WHILE RP @ 1- RP ! REPEAT DROP ;
  7 : READ @ VALUES @ ;
  8 : PUT @ VALUES ! ;
  9 : EXCH LP READ RP READ LP PUT RP PUT
 10       LP @ 1+ LP ! RP @ 1- RP ! ;
 11 : SORT BEGIN DUP DUP LEFT RIGHT
 12   LP @ RP @ > DUP
 13  IF ELSE EXCH THEN UNTIL DROP ;
 14 : COMPARAND OVER OVER LP ! RP !
 15        + 2/ VALUES @ ;  -->
OK
```

Fig. 10.13. Screen 26.

```
SCR # 27       1B H
  0 ( SORTING - PART 5)
  1 R: QUICK COMPARAND SORT
  2    FROM @ RP @ <
  3    IF RP @ DUP END ! FROM @ QUICK
  4    THEN
  5    LP @ END @ <
  6    IF END @ LP @ DUP FROM ! QUICK
  7    THEN R;
  8 : QUICKSORT OVER OVER FROM ! END !
  9   QUICK ;
 10
 11
 12
 13
 14
 15
```

Fig. 10.14. Screen 27.

LEFT and RIGHT are routines to move the left pointer along the array as described above. RIGHT does the same thing for the right pointer. These routines contain WHILE, which makes it possible for no change of the pointer to occur if it is already in an acceptable position.

READ is used to read a value from a location that is pointed at. It needs the address of the pointer on the stack, so is used after the pointer's name, e.g. 'LP READ'. The value stored in that location is left on the stack.

PUT has the reverse action. It is used after the name of the pointer e.g. 'RP PUT'. It places the value which is at second-on-stack into the location in VALUES pointed at by the pointer. READ and PUT are used in EXCH. This requires no values from the stack. It READs the values from the two locations pointed at by LP and RP, then PUTs them back in the opposite locations. In other words, it executes a swap. It follows this by incrementing LP and decrementing RP.

SORT carries out the main sorting action as described earlier. It requires the value of the comparand on the stack. It duplicates this twice, for use by LEFT and RIGHT. These move the pointers to their correct positions. If LP is not greater than RP (LP<=RP), EXCH is called upon to perform the exchange of values. Note that FORTH does not provide a <= operator as such, as does BASIC. Instead, we test to see if LP>RP, and put the required action in the ELSE branch of the IF...ELSE...THEN routine. There is no action in the case of IF being true.

COMPARAND is used to decide on the comparand. It requires the END and FROM values on the stack. Then it calculates the location of the 'middle' of the array or subarray and places the value held in that location on the stack.

QUICK puts most of the above words together. This routine has to call itself to deal with the subarrays into which it divides the main array. It is a recursive routine and must be defined within 'R-colon' and 'R-semicolon' as explained in Chapter Seven. It needs END and FROM on the stack. Then it uses COMPARAND and SORT to perform the sorting of the initial array. After this it has two routines to test the positions of the pointers with respect to the two ends of the array. If one of the two conditions described earlier is met, the value of either END or FROM is amended, so as to define a subarray. Then QUICK calls itself. If neither of these conditions is met, sorting is complete and the word ends.

It needs only one more word, QUICKSORT to take the two location values from the stack and store them in FROM and END. Then QUICK is called upon to perform the sort.

Investigate the way in which QUICKSORT works by testing the action of individual words. It is interesting to redefine EXCH with 'SHOWVALS LP @ . RP @ . FROM @ . END @ .' at the end of it.

This will display the array and the key variables after each swap is done.

QUICKSORT works extremely fast. It was tested by setting up a special array of 101 values, 0 to 100 in reverse (descending) order. Sorting these into ascending order took EXSORT about 20 seconds. A BASIC version of Quicksort took 3½ seconds. QUICKSORT took less than 2 seconds.

To summarise

In this chapter you have found out how to:

- Sort numbers, using Exchange Sort, Bubble Sort and Quicksort.
- Use a BEGIN...WHILE...AGAIN loop.

You have used these FORTH words:

- WHILE causes the action following it to be performed provided it finds a true flag on the stack (flag ...).
- REPEAT used at the end of the action which follows WHILE to make the computer repeat from BEGIN (...).
- 2DROP which drops one double-precision value or two single-precision values from the stack (n ...) or (n1 n2 ...).

Explore more

(1) Adapt one of the sorting routines to sort numbers into reverse (descending) order.

(2) Write a word MEDIAN which, given a set of unsorted numbers, sorts them, and then finds the 'middle number' or median.

(3) Write a word to sort words into alphabetical order, using the ASCII codes of the letters. Hints: (a) Define a word to define an array to hold the words (each word could be up to, say, 10 letters long). (b) Write a word to compare two words letter-by-letter. It starts by comparing the first letter of each, then the second and so on. As soon as it finds that the words are in the wrong order it stops comparing them and swaps them.

Chapter Eleven
Kinds of Numbers

The main kinds of numbers that FORTH uses are:

Number	Range	Number of bytes
Byte	0 to 255	1
Single-precision	-32768 to 32767	2
Double-precision	-2147483648 to 2147483647	4

We have used bytes quite a lot, particularly when dealing with strings. The ASCII codes for the characters are stored as single bytes. There are no ASCII codes greater than 127, so a byte is adequate for this purpose.

We have also used single-precision numbers, noting that these are always integers. The fact that no decimal places are allowed might seem to be a disadvantage. However, we have reached almost the end of the book without needing them, so they cannot be of very great importance. On the other hand, the use of integers is advantageous. It saves memory space and allows the number-handling words to work much more quickly. On those occasions on which it is essential to have decimal places, we can write words to do the job. There is more about this later in the chapter.

The range of single-precision numbers is adequate for most purposes, though sometimes we find that the upper limit is a little too low. For example, we might want to perform this addition:

```
30000 20000 + .
-15536 OK
```

The answer is obviously wrong. The reason for this is that, as explained in Chapter Five, the computer is using 15 bits of the 2 bytes for the value of the number, and the 16th bit for its sign. Adding these two numbers together has carried over into the 16th bit. It becomes a '1',

so the total is taken to be a negative number. The other 15 bits of the total have the value, 15536, so the computer displays - 15536.

Numbers such as those used above are more precisely described as *signed* single-precision integers. The advantage of using signed integers is that we can have negative values as well as positive ones. The disadvantage is that the range is limited to the amount which can be expressed in 15 bits.

If we are prepared to sacrifice the negative numbers, all 16 bits can be used to store the value of a number. This gives what is called an *unsigned* single-precision integer:

```
30000 20000 + U.
50000 OK
```

The addition has taken exactly the same course as before. The difference comes when the computer goes to the top two bytes of the stack and works out what number it represents. 'U-dot' makes the computer take all 16-bits as part of the value. It follows that this word should not be used following calculations which could possibly produce a negative result.

The range of unsigned single-precision numbers is from 0 to 65535.

Double-precision numbers

Really large values are expressed as double-precision numbers. To go with these we have a special range of double-precision words, including:

D+ to add two double-precision words and give a double-precision sum.
D. displays a double-precision value from top-of-stack.
D< compares two double-precision numbers and leaves a true flag on the stack if second-on-stack is less than top-of-stack.
2DROP, 2DUP, 2OVER and 2SWAP are the stack operators for handling double-precision numbers.

These words have just the same action as their single-precision equivalents, so we need say any more about them.

Before a number can be operated on by any of the double-precision words, it must first of all be placed on the stack as a double-precision number, occupying 4 bytes. We may want to use small numbers such as 3, or 22 in double-precision calculations but, if we

simply type them as we usually do, they will be stacked in single-precision format.

The way to indicate that a number is to be stacked in double-precision format is to terminate it with a 'point':

```
12. 400000. D+ D.
400012 OK
```

This may lead to confusion, for there are versions of FORTH in which the terminal 'point' is taken to indicate a floating point number. Consult the reference manual of your version of FORTH if in doubt.

As well as those words which operate entirely with double-precision numbers there are some which involve double-precision numbers at certain stages. The general effect is to maintain precision in the result, which might be lost if single-precision numbers were used throughout. In Chapter Five we came across 'times-divide', in which the product of two single-precision is stored as a double-precision number before being divided by a third single-precision number.

Here are some examples to show how some of these mixed-precision operators may be used:

```
6000 7000 M* D.
42000000 OK
```

'M-times' multiples two single-precision numbers together, giving their product in double-precision format. All values are signed.

```
20000 25000 + 9 U* D.
405000 OK
```

'U-times', in contast to 'M-times' takes all values to be unsigned. It is not possible to key in unsigned numbers directly from the keyboard, so we have had to enter two signed numbers in the normal way. Then they have been added to give a total which is beyond the range of signed numbers. 'U-times' has accepted this as an unsigned number and multiplies correctly. Note that both 'M-times' and 'U-times' accept single-precision numbers, giving a double-precision product. These are *mixed-precision* operators. 'U-times' is the FORTH primitive word for multiplying. It uses machine code, so it is very fast.

There is a primitive for division too:

```
12000139. 3000 U/ . .
4000 139 OK
```

This is another mixed-precision operator. It requires a double-precision number at second-on-stack, to be divided by a single-precision number at top-of-stack. All numbers are taken to be unsigned. The results are expressed as single-precision numbers, the quotient being at top-of-stack, with the remainder at second-on-stack.

Floating-point numbers

Floating-point numbers are those which may have one or more figures after the decimal point. They are not part of standard FORTH, though many versions of FORTH incorporate words for dealing with them. Such versions have words such as F.,F* , F+, and F/ to perform the essential arithmetic operations on them. These words operate in the same way as their single-precision integer counterparts, so there is no point in discussing them further here.

If your version of FORTH has no such words and you need to use floating-point numbers, it is easy to add words to cater for this kind of number.

In practise it is not necessary to devise complicated routines to handle such numbers. The fact that a number is of the floating-point variety is only important when it is to be entered at the keyboard or displayed on the screen. In between it may be stored or operated on by the micro in any form which is convenient to the micro. Indeed, something of this kind already happens. The integers we type in and the integer results we see on the screen are normally in decimal. The micro converts these decimal values to binary form before storing them or operating on them, converting them back again to decimal before it displays the results.

It follows that our main need is for some words to convert floating-point input to integers (single-precision or double-precision) and to convert the integer results back to floating point numbers. In between, the calculations can be handled in the ordinary manner, using integers. We will now develop some words for this purpose for use with versions of FORTH which do not have floating-point facilities.

The words defined in the following paragraphs are useful in a wide variety of applications, though they have some limitations, as mentioned later. First of all, we need a word to accept floating-point numbers from the keyboard. Screen 28 (Fig. 11.1) shows how the word FPIN is defined. The number is to be stored in two parts. The

```
SCR # 28       1C H
  0 ( FLOATING-POINT INPUT )
  1 VARIABLE FIGS   VARIABLE PLACES
  2 : ACC 48 - DUP 1 .R FIGS @ 10 *
  3   + FIGS ! PLACES @ 1+ PLACES ! ;
  4 : CALC DUP 46 = IF 0 PLACES ! EMIT
  5                   ELSE ACC THEN ;
  6 : SLASH DUP 47 = IF ELSE CALC THEN ;
  7 : VALID DUP DUP 45 > SWAP 58 < =
  8   IF SLASH THEN DROP ;
  9 : FPIN 0 FIGS ! 0 PLACES !
 10       BEGIN KEY DUP 13
 11             = NOT WHILE
 12           DUP VALID REPEAT DROP ;
 13
 14
 15
```

Fig. 11.1. Screen 28.

variable FIGS stores an integer which has the same *digits* as the floating-point number but no decimal point, while the variable PLACES stores the number of decimal places in the floating-point number. For example, if the number is 123.45, FIGS holds 12345, and PLACES holds 2. FIGS and PLACES are intended to hold the results of the conversion but not necessarily to store the results permanently. If the application deals with several floating-point numbers, as is likely, the values in FIGS and PLACES are to be transferred to other variables or to arrays. Figure 11.2 shows how the conversion works. There are two ways in which the conversion can be tackled. The user could type all the characters of the number at the keyboard and these would then be held as a string in the input buffer. The conversion could then be carried out on the contents of the buffer, perhaps after it had been transferred to the word buffer. The other approach is to analyse each key-press as it is made. This is the method adopted here. Each key-press is examined and is accepted only if it is a numerical character, a decimal point or a RETURN (indicating that the number is complete). The whole application is included in a BEGIN ... WHILE loop which accepts input until a carriage-return is detected. The 13 on line 10 of Screen 28 (Fig. 11.1) refers to the ASCII code for carriage return, which is 13.

If the key pressed is not the RETURN, VALID checks that one of the other acceptable keys has been pressed. Since the ASCII code for

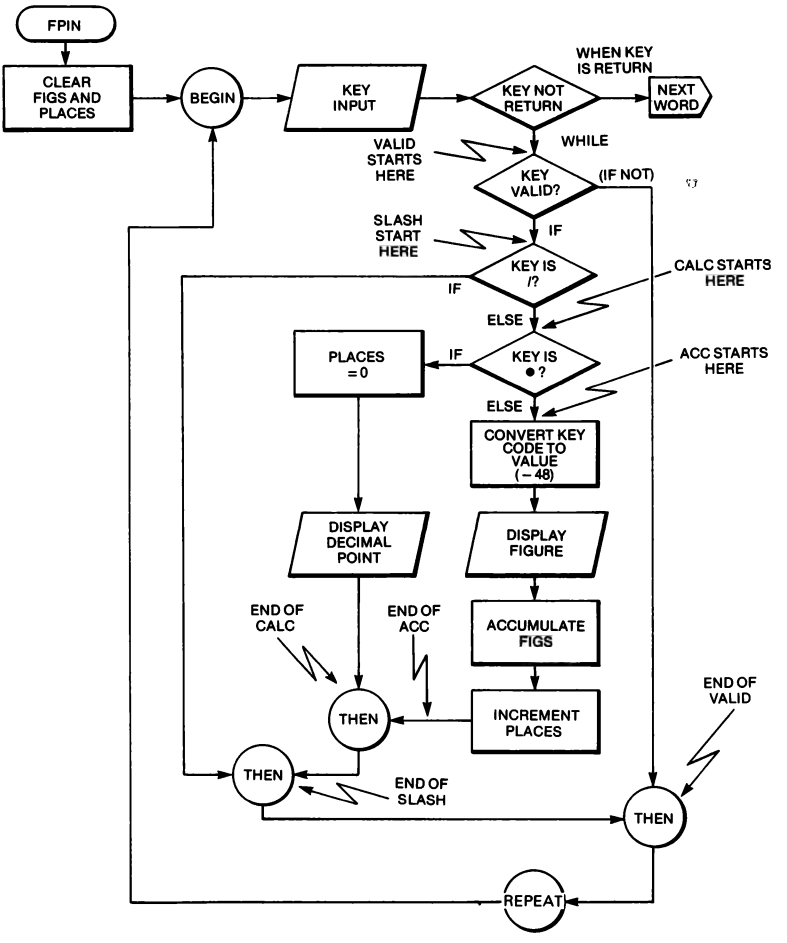


Fig. 11.2. Flow-chart of the floating-point input word, FPIN.

'.' is 46 and the codes for the numerals run from 48 to 57, it is simpler to accept *all* codes in the range 46 to 57 at this stage and reject code 47 (/) later. Code 47 is rejected by the word SLASH. This takes the micro to CALC where the calculation begins. The first step is to detect if the key is '.'. If so, PLACES is set to zero and the decimal point is displayed on the screen. If a numeric key has been pressed, the action passes to ACC which accumulates the figures of the number in FIGS. The code is converted to the actual number by substracting 48. This is displayed on the screen. The effect, as far as the user is concerned, is that the numbers or the decimal point appear on the screen just as they do when being entered in the ordinary way. The figures are accumulated in FIGS by taking the value already in FIGS (see line 2 of Screen 28 (Fig. 11.1)), multiplying

it by 10, adding the current figure to it and storing the result in FIGS. The final step is to increment PLACES, to count the number of figures that has been typed in. CALC resets PLACES to zero when a decimal point is encountered. PLACES thus holds the number of figures entered *after* the decimal point. One thing to be considered is that if the user keys in an integer (for example, 456), PLACES will hold 3. Yet 456 has no decimal places. It is essential that the user should type a decimal point at the end of the number when working with floating-point numbers (for example, 456.). Some FORTHS with floating-point facilities require this terminal point, as already mentioned. In Fig. 11.3 we see FPIN in action.

```
FPIN
123.45OK
FIGS @ . PLACES @ .
12345 2 OK

FPIN
30.006OK
FIGS @ . PLACES @ .
30006 3 OK
```

Fig. 11.3.

The limitation of FPIN depends on the fact that the figures are stored in FIGS, which holds a signed integer. FPIN can accept no more than 5 digits in total and the maximum values it can accept are:

No. of decimal places	Maximum value accepted
0	32767
1	3276.7
2	327.67
3	32.767
4	3.2767
5	0.32767

Numbers converted by FPIN can be used in calculations provided that the following rules are applied:

Addition or subtraction: If the 'places' are the same, 'just add the 'figures'. If the places are different, take the number with the smaller 'places', multiply the 'figures' by 10, and add 1 to the 'places'. Repeat until 'places' is the same for both, then add or subtract the 'figures'.

Multiplication: Multiply the 'figures' together. Add the 'places' together.

Division: Divide one 'figures' by the other. Subtract the 'places' of the divisor from the 'places' of the dividend. If 'places' becomes negative, multiply the resulting 'figures' by 10 and increment 'places' by 1. Repeat until 'places' becomes zero.

Take care that operations such as these do not make 'figures' or 'places' greater than the maximum allowed. It is easy to see that handling floating-point numbers rapidly becomes rather complicated, which is why they are best avoided as far as possible.

Screen 29 (Fig. 11.4) shows words for taking floating-point numbers, stored in FIGS and PLACES, and displaying them on the screen.

```
SCR # 29        1D H
  0 ( FLOATING-POINT OUTPUT )
  1 : DIVIS 1 BEGIN PLACES @ DUP 0>
  2             WHILE SWAP 10 * SWAP 1-
  3                   PLACES !
  4             REPEAT DROP ;
  5 : INTEG DUP FIGS @ SWAP /MOD 5 .R ;
  6 : ZEROES BEGIN OVER OVER <
  7            WHILE 10 / ." 0" REPEAT ;
  8 : DPLS SWAP DUP 0=
  9          IF 2DROP
 10        ELSE ." ." 10 / ZEROES
 11             DROP . THEN ;
 12 : FPOUT DIVIS INTEG DPLS ;
 13
 14
 15
```

Fig. 11.4. Screen 29.

DIVIS calculates the value of a divisor which is to be left on the stack for use by INTEG. This divisor has to have a value 10 to the power of the value in PLACES:

Value in PLACES	Value of divisor
0	1
1	10
2	100
3	1000
4	10000
5	100000

DIVIS is a BEGIN...WHILE loop which multiplies the value on the stack (initially 1) by 10 and decrements PLACES, until PLACES has been reduced to zero. INTEG finds this divisor on the stack. Its subsequent action is (top-of-stack to right):

INTEG	divisor		
DUP	divisor	divisor	
FIGS @	divisor	divisor	FIGS
SWAP	divisor	FIGS	divisor
/MOD	divisor	remainder quotient	
5 .R	divisor	remainder	

'Dot-R' displays quotient on the screen, so giving us the figures before the decimal point. For example if FIGS is 12345 and PLACES is 2, the divisor is 100. 'Divide-mod' gives a quotient of 123 and a remainder of 45. The screen displays 123, and the 45 is still on the stack, ready for the decimal places to be displayed by DPLS.

DPLS uses a word ZEROES to work out if there are any zeros to be displayed before the non-zero figures. It might happen, for example, that the remainder is 5, yet there have to be *two* decimal places, to give .05. ZEROES repeatedly divides the divisor (left at second-on-stack by INTEG) by 10 and displays a zero each time, until the divisor has become less than the remainder. This produces the right number of leading zeroes.

DPLS begins by checking the remainder; if there is none, the action ends. If there is a remainder, it displays the decimal point. DPLS then calls on ZEROES to display any zeroes that may be required. Finally it displays the remainder to complete the figures after the decimal point. The word FPOUT combines the actions of these words into one sequence (See Fig. 11.5).

```
FPIN
33.440K
FPOUT
    33.44 OK
```

Fig. 11.5.

Some suggestions for adding to FPIN and FPOUT appear at the end of this chapter, under EXPLORE MORE.

Other number systems

FORTH has several variables of its own which are automatically set

to certain values when the computer is switched on. The one we are interested in in this section is BASE:

```
BASE @ .
10 OK
```

The initial value in BASE is 10. This is the base of the number system that FORTH normally uses, the decimal system. The value in BASE can be changed:

```
8 BASE !
OK
```

See what effect this has on a simple addition and multiplication:

```
6 5 + .
13 OK

6 4 * .
30 OK
```

Obviously the micro is no longer working in the decimal system. It is now working in octal, or base-eight, scale. The displayed answer does not mean 'thirteen' as we would understand it in decimal. The '1' now represents 8 instead of 10. So '13' is interpreted as $1\times8+3=11$, in the decimal scale. This is the expected answer to the addition. The answer to the multiplication is interpreted as $3\times8+0=24$, in decimal. This too is as expected.

The binary scale (base-two) is frequently used in computing:

```
2 BASE !
OK
```

Here are a binary addition and multiplication:

```
1 1 + .
10 OK

10 11 * .
110 OK
```

1 plus 1 gives 2 in decimal, but the computer is working in binary, so it displays the result as '10'. This is not 'ten', but $1\times2+0=2$. In the multiplication we have '10' (2 in decimal) multiplied by '11' (3 in decimal), giving '110' (6 in decimal).

Try some other simple calculations in binary, and check that the computer gives the correct binary answer. To find out the decimal

equivalent of a binary number, all you have to do is to place it on the stack, and then instruct the computer to revert to working in decimal. We could put 10 back into BASE by using '10 BASE !', but the word DECIMAL is more convenient:

```
110 DECIMAL .
6 OK
```

The value 110 placed on the stack is now displayed in decimal form. Of course, the computer is actually working in binary all the time. Numbers stored are stored on the stack in binary and all the calculations are in binary. Changing the base to 8, 10 or any other value simply tells the micro which base *we* want to work in. It then knows that we want it to accept our input with reference to the selected base. For example, if we type '100' when BASE is 2, it stores it directly as '100'. But if BASE is 10, it knows we mean 'a hundred', and stores the value as the binary equivalent of a hundred, 1100100. Similarly, when displaying a number on the screen, it displays it in whatever base we have selected. Here is a word to make the computer display in decimal a value which we have entered in binary:

```
: BIN DECIMAL . 2 BASE ! ;
```

This word is very useful when you are working out the values for user-defined graphics, as in Chapter Six. First put the computer into binary mode by typing '2 BASE !'. Then use BIN, as below:

```
11010110 BIN
214 OK
```

The binary number is made up by looking along one row of squares of the 8 × 8 grid of the character. For each shaded square type 1, for each unshaded square, type 0. The computer accepts the resulting 8-bit binary number and puts it on the stack. Then the word DECIMAL (in BIN) restores the computer to decimal operation to display top-of-stack in decimal. In this example, the binary number is equivalent to 214 in decimal. This is the value to be used in defining the graphics character. BIN ends by putting the computer back into binary mode, ready for you to enter the details of the next row of the grid.

HEX is another base-changing word which, as might be guessed, makes the computer receive input and display output in hexa-decimal. Figure 11.6 shows it in action. In the first example, 6 plus 5

```
HEX
6 5 + .
B OK
3 5 * .
F OK
F 1 + .
10 OK
```

Fig. 11.6.

gives eleven (decimal) which is displayed as B, its hexadecimal equivalent. In the second example, three fives give fifteen, displayed as F. Hexadecimal uses the numerals 0 to 9 and letters A to F. The third example shows what happens next. Adding 1 to F gives 10, the hexadecimal equivalent of sixteen.

Figure 11.7 gives HD, which converts hexadecimal numbers to decimal numbers. FORTH does not limit us to the 'popular' number

```
: HD DECIMAL U. HEX ;

FFEE HD
65518 OK
```

Fig. 11.7.

bases. You can work in any of a wide range of bases. Figure 11.8 shows what happens in base-70. Obviously base-70 requires 70 different characters to express its numbers. It uses the numerals 0 to 9, then it will need all the letters of the alphabet. This provides only 36

```
70 BASE !
OK

Z 1 + .
[ OK


Z Z * .
HZ OK
```

Fig. 11.8.

characters to use so far. The first example above shows what happens next. The base 70 equivalent of the decimal number 37 is the left-hand square bracket. You may find that it uses a different symbol on your micro. Z squared gives HZ. In order to check on this

we had better look at the entire range of symbols. Figure 11.9 shows how the word BASE70 displays these.

```
: BASE70 11 0 DO I . LOOP ;
OK


BASE70
0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J
K L M N O P Q R S T U V W X Y Z [ \ ] ↑
← ↓ a b c d e f g h i j k l m n o p q r
s t u v w x y z { | 10 OK
```

Fig. 11.9.

Remember that the computer is in base 70 mode when it executes the loop, so that the values 11 and 0 before DO are equivalent to 71 and 0 in decimal. Once again, the exact symbols shown in your version of FORTH may differ from those in the printout. Now we can see that H is equivalent to 18 in decimal, while Z is equivalent to 36. The second of the examples given above calculates Z-squared (which is 36×36=1296). The result is interpreted as 18×70+36=1296. So Z×Z=HZ is a correct base-70 multiplication.

Random numbers

One of the best-known ways of obtaining a random number is to throw a dice. This is the basis of so many games in which chance is intended to be the main element. Provided that the dice is not a biased one and that it is fairly thrown, there is no way to tell in advance which of its six faces will be uppermost when it comes to rest. The result of the throw may be any number between 1 and 6, and each of these numbers is equally likely to be thrown on any one occasion. The result of the throw is a *random number.*

If you want to obtain random numbers in other ranges, you can use special polyhedral dice. Some of these have 20 faces, giving you random numbers in the range 1 to 20. Another way of getting random numbers is to cut a pack of playing cards.

Random numbers play an important part in many computer games, so we need a way to make the micro produce them. This is not strictly possible, since a computer is a piece of machinery with

exactly predictable behaviour (some programmers may not believe this!). There is no action in a computer which has the chance element of rolling a dice. Instead we generate *pseudo-random numbers,* which, from now on, we will refer to simply as random numbers, remembering that they are not truly random. The computer is programmed to generate a series of numbers in such a way that successive numbers *appear to be* produced at random. The way they are calculated is not apparent to the user, so it is virtually impossible to predict which number will appear next. The series of numbers is a very long one, which will repeat itself eventually, but not until hundreds of numbers have been generated.

The usual equation for generating these numbers is:

$$\begin{pmatrix}\text{Random}\\ \text{number}\end{pmatrix} = \begin{pmatrix}\text{Previous}\\ \text{random number}\end{pmatrix} \times A + C \quad \begin{pmatrix}\text{modulo}\\ M\end{pmatrix}$$

A, C and M are constants. M is very large, being perhaps the largest number that the computer can store. 'Modulo M' means that every time the calculation gives a random number exceeding M, it is reduced by M, so keeping it within the range 0 to M. A and C can have almost any values, the main point being that A, C, and M have no factors in common. In the words defined below, M is 32767, the value of the largest possible signed integer. A is 2011 and C is 5. Since both of these are prime numbers they have no factors in common with each other or with M.

To start off the generation of random numbers we need a 'seed'. This can be given any value and, in effect, represents the previous random number in the equation above. After each random number has been generated it is stored in SEED ready for generating the next number. Figure 11.10 gives the words we need. RND1 performs the calculation given in the equation above. It takes the previous number from SEED, multiplies it by 2011 and adds 5. The result is ANDed with 32767. The way this works is explained in Chapter Twelve. Its effect is to remove any '1' which may have appeared in

```
VARIABLE SEED
OK
: RND1 SEED @ 2011 * 5 +
       32767 AND DUP SEED ! ;
OK
: RND RND1 32767 */ ;
OK
```

Fig. 11.10.

the 16th digit, so keeping the value of the number within the range of positive signed integers (0 to 32767). The new value is stood in SEED and a copy of it is left at top-of-stack.

RND1 has produced a random number in the range 0 to 32767. But to simulate the throwing of a dice, for example, this range is far too great. RND allows us to specify the range. RND requires you to put a value on the stack to indicate the maximum value the random should have. 'Times-divide' then multiplies the random number (left by RND1) by your maximum value, and divides the product by 32767. This produces an integer in the required range. Since division rounds *down* in FORTH, the maximum value that the random number can have is one less than the value you have placed on the stack.

Figure 11.11 shows a word to simulate the throw of a six-faced dice. Rather than have just one throw we use a loop to 'throw' it 100

```
: DICE 100 0 DO 6 RND 1+ . LOOP ;


DICE
1 5 2 5 4 4 3 4 5 3 4 4 6 6 5 1 6 1 5 2
3 5 4 6 3 2 3 6 5 1 3 6 2 3 6 6 6 6 4 3
4 5 3 2 6 2 3 4 6 2 1 1 4 6 4 4 5 4 5 5
5 3 5 6 1 2 2 4 1 5 6 3 5 2 5 1 6 1 4 2
5 3 4 4 4 4 4 1 4 4 6 4 3 4 1 1 1 2 3 3
OK
```

Fig. 11.11.

```
: CRAPS 100 0 DO 6 RND 1+
                6 RND 1+
                + .
           LOOP ;


CRAPS

9 8 6 7 8 4 5 10 10 11 7 8 9 6 5 10 10 7
6 10 5 7 5 9 6 9 9 5 5 7 7 9 4 3 10 9 9
 9 3 5 3 6 5 6 4 2 8 7 7 10 9 7 8 11 6 7
 7 6 9 8 4 8 9 2 5 9 6 7 8 4 5 7 8 11 9
6 7 7 8 7 5 12 6 9 7 3 3 9 2 8 12 10 4 4
 5 6 6 5 11 7 OK
```

times. Then we shall be able to check that it is a 'fair' dice.

There is one problem. Dice are numbered from 1 to 6, but RND produces values ranging upward from *zero*. The word DICE adds 1 to each random number so that the range begins from 1. This being so, we need RND to produce numbers in the range 0 to 5. The maximum value placed on the stack is therefore 6.

If you check through the printout, you will find that all 6 possible numbers occur in approximately equal proportions. The dice is fair!

Figure 11.12 shows a word to simulate the throwing of two dice together, as in the game of Craps. Each loop does two 'throws' and the results are added before displaying their total. The printout shows that the most frequent result is 7, while the lowest and highest possible numbers, 2 and 12, occur very rarely.

Learn your tables

Even in these days of computers and pocket calculators, ability at mental arithmetic is important. The remainder of this chapter describes a FORTH application which gives practice at the multiplication tables from 1 to 10. PAIR selects two random numbers in the range 1 to 10 and places them on the stack (See Fig. 11.13).

```
: PAIR 9 RND 1+ DUP 9 RND 1+ DUP ;
OK
: SHOW ROT . ." TIMES " . ." = " ;
OK
: ASK * DUP QUERY 32 WORD NUMBER DROP ;
OK

: CHECK = IF ." CORRECT" DROP
          ELSE ." WRONG , IT MAKES " .
          THEN CR ;
OK
: MULTIPLY BEGIN PAIR SHOW ASK CHECK
           AGAIN ;
OK
```

Fig. 11.13.

SHOW displays these on the screen as a multiplication problem, leaving the answer blank. ASK accepts input from the user, who is to type in the answer to the problem, then press RETURN. ASK first multiplies the two numbers together to obtain the correct answer on top-of-stack. QUERY accepts the string of characters (the figures

typed by the user) into the input buffer. WORD transfers this to the word-buffer with a space as delimiter. NUMBER converts this into a double-precision value, placed on the stack. Since we are dealing only in single-precision values, DROP gets rid of the top two bytes, which each hold only zero. This leaves the users's answer on top-of-stack with the computer's answer below it.

CHECK compares the two answers. If they are equal, it displays 'CORRECT'. If not, it states that the answer is wrong and displays the correct answer. A carriage return follows, so that the next problem is displayed on the line below. MULTIPLY has all these words inside a BEGIN ... AGAIN loop, so that an endless series of problems is presented. Figure 11.14 shows how it runs. As can be seen, the way to end the inquisition is by pressing ESCAPE

```
MULTIPLY
8 TIMES 7 = 56
CORRECT
1 TIMES 3 = 77
WRONG , IT MAKES 3
1 TIMES 8 = 8
CORRECT
7 TIMES 7 = 77
WRONG , IT MAKES 49
3 TIMES 2 =
Escape
OK
```

Fig. 11.14.

To summarise

In this chapter you have found out how to:

- Use numbers of different types.
- Work in different number bases.
- Generate random numbers.

You have used these FORTH words:

- U. 'U-dot' displays a single-precision integer in unsigned form (n ...).
- D+ 'D-plus' adds two double-precision numbers to give a double-precision sum (n1\n2 ... n1+n2).
- D. 'D-dot' displays a double-precision number (n...).

- M* 'M-times' multiplies two signed single-precision numbers giving a signed double-precision product (n1\n2 ... n1*n2).
- U*'U-times' as M* but uses unsigned numbers (n1\n2 ... n1*n2).
- U/ 'U-divide' divides a double-precision number (n1) by a single-precision number (n2), leaving a single-precision remainder and quotient on the stack. All numbers are unsigned (n1\n2 ... remainder\quotient).
- BASE a variable storing the current number base being used by the computer (...).
- DECIMAL gives decimal input and output (...).
- HEX gives hexadecimal input and output (...).
- NUMBER converts a character string into a signed double-precision number. It requires an address at which the number of bytes in the string is stored, and assumes the string itself is stored in the addresses immediately after this (address ... n).

You have learned that:

- Numbers are stored as bytes, as single-precision numbers or as double-precision numbers.
- Numbers can be signed or unsigned.
- There are single-precision, double-precision, and mixed-precision operators.
- The computer can be programmed to generate pseudo-random numbers.

Explore more

(1) The only double-precision conditional operator normally provided in FORTH is 'D-greater'. Write your own version of the words 'D-less' and 'D-equals'.

(2) Revise FPOUT so that there is no need for the user to key a decimal point when there are no decimal places in the number.

(3) Revise FPIN and FPOUT so that FIGS can hold a double-precision value, so allowing the range of acceptable floating-point numbers to be increased.

(4) Write some words for floating-point addition and other arithmetical operations. Descriptions of how to perform these operations were given earlier in the chapter in a form that is easy to translate into FORTH.

(5) Write a word DH which lets the user type in a decimal number and then displays it in hexadecimal.

(6) Write adaptations of MULTIPLY to test the user's ability to add, subtract, or divide.

(7) Write the words for this simple game. Use RND to place 10 aliens (user-defined characters) at randomly chosen positions on the screen. Then steer your spacecraft (another special character) around the screen, using four keys to direct it. When your craft 'runs over' an alien, the alien disappears from the screen. The aim is to destroy all the aliens with the least number of changes of direction (least number of key-presses), so you need to plan your route very carefully.

Chapter Twelve
AND and OR

This chapter deals with the way in which FORTH performs logical operations. There are two main kinds of logical operation, one of which we have met already and have used a lot. Flag logic, as one might call it, requires a flag value to be left on the stack. A flag is a value, the main purpose of which is to signal to the computer that a condition is true or false. There are a number of words which look at the value on top-of-stack, treating it as a flag and acting accordingly, The flag is taken to mean 'false' or 'true' according to whether its value is zero or not. Words which use flags in this way are:

IF
UNTIL
WHILE
?DUP

The last in the list is a version of DUP which acts only if the top-of-stack is a true flag. In other words, if the top-of-stack is zero, then zero is left at top-of-stack. If it is some other value, it is duplicated. Note this word does not remove the flag from the stack, whereas the other words do.

A flag can be placed on the stack either as the result of a calculation or by the action of a conditional operator. The conditional operators that we have used are:

=	0=
>	0>
<	0<

These leave 0 on the stack to indicate 'false' and 1 to indicate true. All of the words mentioned above are concerned with the truth, or otherwise, of given conditions and enable the computer to take decisions based on logic.

Another logical word that we have used several times is NOT.

This reverses a flag. If the flag is true, it changes it to false. If it is false, it changes it to true. Its action is exactly the same as 0=. Whether we use NOT or 0= is mainly a matter of preference. Sometimes it may make the logic of the program clearer to the reader to use one rather than the other.

The other kind of logical operation uses the two words AND and OR. It is these which are the main topic of this chapter. We have already used AND a few times in previous chapters. It is time to explain the way it works.

Readers who have used BASIC will probably have come across AND and OR already. In BASIC, they can be used in two rather different ways. Most versions of BASIC use AND and OR as *logical operators,* as in the statement:

```
IF J = 3 AND K = 6 THEN X = X + 1
```

The AND links together two conditions (J = 3, K = 6), *both* of which have to be true if the action after THEN is to be performed. OR is used similarly.

The other way of using AND and OR in BASIC is entirely different. The simpler BASICS do not allow them to be used in this way. This is sometimes referred to as 'bit-by-bit' or 'bitwise' operation. When used in this way, AND and OR are described as *Boolean operators.* FORTH uses AND and OR as Boolean operators, not as logical operators. From now on the discussion refers only to this kind of operation.

Here is AND in action as a Boolean operator (we will come back to the 'bitwise' aspect in a moment!):

```
12 10 AND .
8 OK
```

Obviously it is capable of operating on numbers, but the results it produces do not conform to any ordinary arithmetical operation. Here is OR at work:

```
12 10 OR .
14 OK
```

Once again, it is doing something with the numbers, but exactly what it is doing is not clear. The term 'bitwise' gives us a clue. AND and OR are operating on the *bits* (the binary digits) of the numbers, rather than on the numbers as a whole. To follow what is happening, we need to put the micro into binary (base-2) mode:

```
2 BASE !
OK
```

Here is the same operation (10 AND 12) but keyed in and displayed in binary:

```
1100 1010 AND .
1000 OK
```

The result is equivalent to 8 in decimal, the same result as before.

It helps the explanation if we set out the two original values one below the other, so that the bits of one come directly below the corresponding bits of the other:

```
      12 is  1 1 0 0
      10 is  1 0 1 0

Result  8 is  1 0 0 0
```

AND works by placing a 1 in the result only if the corresponding bit in one number AND the corresponding bit in the other number are 1s. Only the left-most bits of both numbers are 1s, so only the left-most bit of the result is a 1. A pair of 0s or a 1 and a 0 produce only a 0. Try keying in a few more numbers in binary and see what AND does to them, as in the example above.

Here is the action of OR.

```
1100 1010 OR .
1110 OK
```

Setting the numbers out as before:

```
      12 is  1 1 0 0
      10 is  1 0 1 0

Result 14 is  1 1 1 0
```

OR works by placing a 1 in the result if the corresponding bit in one number OR the corresponding bit in the other number is a 1. It is only when *neither* of the bits is a 1 that the result contains a 0. In the example above only the right-most bit has zeros in both numbers, giving a zero in the result. Experiment with OR, using other pairs of binary numbers.

Shifting

Now that we are treating bits as individuals, there are several other operations that we can perform on them. One of these is known as

shifting. The idea is that we begin with a number of bits (for example, a byte of 8 bits) which might be like this:

0 0 0 1 0 1 0 0

A left shift moves each bit one place to the left:

0 0 1 0 1 0 0 0

Successive left shifts give:

0 1 0 1 0 0 0 0
1 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

At each shift the left-most bit is lost, and the right-most bit becomes a zero. The right-shift operation works in the opposite direction.

It might seem to be a complicated matter to perform a left-shift, but in practice it is extremely easy. All that has to be done is to multiply the binary number by 2! Use LS to demonstrate this:

```
: LS DUP U. 10 * ;
```

Remember that the micro has to be in binary mode. In LS the figure 10 represents decimal 2. 'U-dot' is used to display the result so that we can see all 16 bits of the number. LS leaves the shifted number on the stack, so we can see the effect of repeated left-shifts. We start with any binary number (See Fig. 12.1). The leading zeros are not

```
10100 LS
10100 OK

LS
101000 OK

LS
1010000 OK

LS
10100000 OK
```

Fig. 12.1.

displayed, of course, but it is clear that the left-shift is working as required. Try some more left-shifts to see what happens as each 1 becomes the 16th (left-most) bit.

Bit maps

One important application of bitwise logic is in the storage of data. Normally we store data in a computer as a set of numbers, which may represent actual quantities or may be other kinds of information in coded form. For example, the value 5 stored in a stock-taking program may mean that you have 5 packets of WYTO in stock, or it may mean that this item and all other items coded 5 have special storage requirements. Quite often the codes or the values that we need to store are relatively small numbers. Yet they may each be stored as single-precision integers, requiring 2 bytes for each piece of data. It may easily happen that there is not enough memory to allow us the luxury of storing data in this way. It must be compacted.

One way is to use single bytes for storing data. These can hold values ranging from 0 to 255 which is more than enough for many purposes. As already explained, FORTH provides several words for handling bytes.

If we use the bits themselves for storing individual items of data we can save even more memory. This is what is meant by a *bit map.*

The most compact form of data storage is obtained when we treat the bits as flags. Each bit indicates the truth or otherwise of a given condition. For example, an estate agent might want to set up a data bank about dwellings for sale in the district. The information held in one byte might be as follows:

Bit no.	=0 if	=1 if
0	1-storey	2 or more storeys
1	furnished	unfurnished
2	built pre-war	built post-war
3	completed	under construction
4	no central heating	centrally heated
5	no double-glazing	double-glazing
6	near shops	no shops near
7	no garage	garage

Each house on the register would be represented by a byte in memory, holding this data. Thus an unfurnished pre-war house with central heating, no double-glazing, close to the shops but with no garage is coded as:

0 0 0 1 0 0 1 1

Bits are numbered from right to left.

Here is a very compact way of holding such information. The compactness is not only important from the point of view of memory space, but also for saving disk or tape space, and (with tape) the time required for loading and saving. Moreover, data stored in this way can be scanned very quickly by the computer, and the required piece of information is quickly retrieved from a very large amount of stored data.

It is possible to combine this bit-map method with the storage of numerical data. For example, three of the bits in a byte could be used to hold the number of bedrooms in the dwelling. Three bits cover the range 0 to 7, and '7' can be taken to mean '7 or more bedrooms'. The other bits would be used as described above. A single byte cannot hold all the data that an estate agent might want to store, but it is possible to pack into 3 or 4 bytes all the essential information that an intending house-purchaser might want to know in the first instance. One of the bytes could hold a coded reference to the office file in which complete details, maps and photographs are available.

Identifying by logic

The example of the house agent, shows how we can use computer logic for identification. The house buyer tells the agent which features are most important in selecting the house. The agent then uses the computer to search the stored information to *identify* which house or houses conform to the requirements of the buyer. If the buyer specifies only a few features, the computer may be able to find many dwellings which will satisfy the buyer. If the buyer stipulates a long list of essential features, the computer may find only few or perhaps no dwellings to meet the buyer's demands.

Here is another application of the same idea. Those who are interested in bird life often want to identify the birds they see, but this is not always as simple as it might be. The bird may be a long way away or may be partly hidden by vegetation, so that all of its features cannot be clearly seen. Or it may fly away out of sight before the watcher has had time to notice all its features. The kind of question the computer could answer is: 'I saw a long-legged bird beside the lake. It was white in parts, but the light was too poor to pick out its other colours. What kind of bird could it be?'. The computer then searches the stored bytes, looking for any birds that spend their time near water, have long legs and are at least partly white. It will ignore

other features and pick out all those kinds of birds which conform to the description. It prints a list of all such birds it finds listed in its memory.

Screen 30 (Fig. 12.2) lists nine birds. It is easy to extend the list to cover several screens if required. The screen is being used for storing

```
SCR # 30        1E H
  0 ( NAMES OF BIRDS
  1 SPARROW, HSE: M. BLK. THROAT     64
  2 ROOK: BARE FACE PATCH CF CROW   94
  3 BLACKBIRD: M. YELL. BEAK; F.BWN 10
  4 MARTIN, HSE: WHT RUMP CF SWALLOW 56
  5 SWALLOW: LONGER TAIL THAN MARTIN 54
  6 REDBREAST/ROBIN                  18
  7 HERON, GREY: DARK FLIGHT FTHRS   152
  8 GULL, COMMON: GR-YELL BEAK & LEGS 184
  9 TIT, GT: BLK STRIPE ON BELLY     36
 10
 11
 12
 13
 14
 15    ) -->
```

Fig. 12.2. Screen 30.

text, any line of which can be picked out and displayed. Since the whole screen is text and there are no FORTH words or definitions, the whole screen is enclosed in brackets. The first bracket, '(', at the start of line 0 is in fact a FORTH word, called 'bracket'. This has been used on line 0 of all the other screens shown in this book. Since it is a FORTH word it must be followed by a space. When the computer encounters such a bracket, it ignores what follows until it finds a reversed bracket, ')', which acts as a delimiter. Normally we place the delimiting bracket at the end of line 0, so that only the title of the screen is ignored. Here we place the reversed bracket on line 15. The main point to remember when keying in the text is that it must not include brackets. 'Bracket' can be used in word-definitions too, to hold remarks about the action of the word.

Screen 30 (Fig. 12.2) lists the birds and follows their names with brief notes on special features. The house sparrow has a note that the male has a black throat. The rook has a note to the effect that it has a bare face patch which the crow does not have (CF stands for compare with). These notes help the user to confirm the identification, by quoting features that help distinguish the bird in question from other birds of similar appearance. The numbers at the end of each line are references to pages in a bird book

on which detailed descriptions and pictures are to be found. By referring to these the user should be able to decide which of several listed birds was the one seen.

Screen 31 (Fig. 12.3) is a list of eight features which could be used for identifying birds. The contents of this screen are to be displayed

```
SCR # 31       1F H
  0  (  BIRD FEATURES
  1  1   IS LESS THAN 20 CM LONG
  2  2   HAS BLACK FEATHERS
  3  3   HAS WHITE FEATHERS
  4  4   HAS BROWN FEATHERS
  5  5   HAS RED FEATHERS
  6  6   SPENDS TIME ON THE GROUND
  7  7   A WATER OR WATER-SIDE BIRD
  8  8   NESTS IN TREES, NOT BUSHES
  9
 10
 11
 12
 13
 14
 15   )   -->
```

Fig. 12.3. Screen 31.

so that the user can key in the numbers corresponding to the features of the bird. These features are of several kinds. There are visual features such as overall size (line 1), and feather colours (lines 2 to 5). Certain birds frequent particular localities so the place the bird is seen at can be a useful identifying clue (lines 6 and 7). The habits of the bird also can be very helpful (line 8). It is not intended that the user shall be able to decide one way or the other about *all* of these features. As explained earlier, the difficulty with bird identification is that one is able to discover only a few features on a given occasion. The user is to key in those features which are positively known.

Screen 31 (Fig. 12.3) covers only eight features, enough for storing in a single byte. It is easy to add many more features, adapting the application to deal with two, three or more bytes for each bird.

The stages of operation of the application are:

(1) Display a numbered list of the features.

(2) The user keys in the numbers corresponding to features seen on the bird.

(3) The computer builds up a byte, called BYTE, in which the bits indicate those features keyed in by the user.

(4) The computer already holds in its memory a set of bytes, one for each bird, in which the bits indicate the features possessed by each bird. It searches these bytes to find those which have bits in common with BYTE. For any it finds it displays the name of the corresponding bird.

Screen 32 (Fig. 12.4) holds all the FORTH words for identifying birds, using the word BIRDS.

```
SCR # 32        20 H
  0 ( IDENTIFYING BIRDS )
  1 CREATE FEATS 175 C, 162 C, 42 C,
  2 7 C, 23 C, 61 C, 198 C, 102 C,
  3 135 C,
  4 VARIABLE BYTE
  5 : BITS 0 DO 2* LOOP 2/
  6              BYTE @ OR BYTE ! ;
  7 : FEATURES 0 BYTE ! BEGIN QUERY 32
  8 WORD  NUMBER DROP DUP 0>  WHILE
  9     1 SWAP BITS REPEAT ;
 10 : GET BYTE @ 9 0 DO DUP DUP I FEATS
 11  + C@ AND = IF I 1+ 30 .LINE CR
 12         THEN LOOP DROP ;
 13 : BIRDS 12 >VDU 9 1 DO I 31 .LINE
 14   CR LOOP CR ." KEY THE NUMBERS:"
 15   CR FEATURES CR ." IT COULD BE:"  CR GET ;
```

Fig. 12.4. Screen 32.

The first thing to be done is to store the data about the birds in memory. Before writing the application we set out a table like this:

Bird	Nests in trees	Wat.	Grnd.	Red	Brn.	Wht.	Blk.	<20	Decimal
Sparrow	1	0	1	0	1	1	1	1	175
Rook	1	0	1	0	0	0	1	0	162
Blkbird	0	0	1	0	1	0	1	0	42
H Martin	0	0	0	0	0	1	1	1	7
Swallow	0	0	0	1	0	1	1	1	23
Redbrst	0	0	1	1	1	1	0	1	61
Heron	1	1	0	0	0	1	1	0	198
C Gull	0	1	1	0	0	1	1	0	102
Gt Tit	1	0	0	0	0	1	1	1	135

The table shows a 1 if the bird possesses the feature given at the head of the column and a 0 if it does not. Note that the *female* blackbird is *brown,* as allowed for in the table. The column headings are in the reverse order to the listing of features in Screen 31 (Fig. 12.3) for reasons which will be apparent later. Each row of the table shows the

composition of the byte which holds the features of each bird. The right-hand column gives the equivalent decimal value of that byte. Now we are ready to put this information into memory.

Line 1 of Screen 32 (Fig. 12.4) shows a new way of using CREATE. This time it is not being used with DOES> to define a defining word. Instead it is being used simply to create the head of a word in the dictionary. The word is FEATS. Its code field contains the address of a routine to put the address of its parameter field on the stack every time FEATS is used. The parameter field of FEATS is filled immediately by using 'C-comma'. This word stores a byte at the next available address in the dictionary. In other words, it places it in the first free byte of the parameter field of FEATS. The values, which you will recognise as the values calculated from the table above are stored in succession on the parameter field of FEATS. This is a simple way of providing an array filled with data.

The variable BYTE is defined next, to hold the details of the bird as the user keys them in.

BITS is the word which puts the bits into BYTES. It requires at second-on-stack the feature number (1 to 8) as keyed in by the user. It also requires 1 at top-of-stack. It goes through a loop from 0 to the feature number, multiplying by 2 each time. We use the word '2-times', a special word for fast multiplication by 2. This gives a shift-left operation, as described in the previous section. For example, if the user keys in 4 (brown feathers), BITS shifts 1 four times:

Start	0 0 0 0 0 0 0 1
1st shift	0 0 0 0 0 0 1 0
2nd shift	0 0 0 0 0 1 0 0
3rd shift	0 0 0 0 1 0 0 0
4th shift	0 0 0 1 0 0 0 0

This has taken the 1 too far. The '2-divide' after the loop shifts it one space to the right again:

Right-shift 0 0 0 0 1 0 0 0

The reason for this is that a DO ... LOOP must always execute at least once. So the 1 is always shifted at least once, to position 2. The '2-divide' is to compensate for this. This again is a special fast-action word. The resulting byte is then combined with the value already in BYTE, using OR. The reason for this is that the user will probably key in several numbers and the corresponding bits are to be stored in BYTE one at a time. For example, if the user has already keyed in 2 and 6, BYTES will hold:

0 0 1 0 0 0 1 0

(bits numbered from right to left, 1 to 8). Now the user keys in 4 as already described. BITS has produced:

	0 0 0 0 1 0 0 0
BYTES holds	0 0 1 0 0 0 1 0
OR these two	0 0 1 0 1 0 1 0

The new bit, positioned by BITS, has taken its place among those already present in BYTES. The bits already there are not affected by this operation.

FEATURES is the word which accepts the numbers typed in by the user. It starts by putting zero into BYTES, clearing it ready for storing the bits. Then a BEGIN ... WHILE ... REPEAT loop accepts numbers, one at a time. The routine used was described in Chapter Eleven. The result is to put the number on top-of-stack. This is duplicated to keep a copy for later action. The top copy is tested to see if it is greater than zero. If so, action continues. If the user has keyed 0, it is taken to indicate that there are no more features to be typed in and FEATURES ends. The action of FEATURES in response to a non-zero number is to put 1 on the stack and swap it with the number. It then calls BITS to position the bit and insert it in BYTE.

GET is the word which searches FEATS comparing each byte there with the value in BYTE. Its stack action is:

GET					
BYTE @	BYTE				
9 0 DO	BYTE				
DUP DUP	BYTE	BYTE	BYTE		
I FEATS	BYTE	BYTE	BYTE	I	FEATS
+	BYTE	BYTE	BYTE	addr	
C @	BYTE	BYTE	BYTE	byte	
AND	BYTE	BYTE	BYTE.	byte	
=	BYTE	flag			
IF	BYTE				
I 1+	BYTE	I+1			
30	BYTE	I+1	30		
. LINE CR	BYTE				
THEN	BYTE				
LOOP	BYTE				
DROP					

In the table above, 'addr' refers to the address of a particular byte in FEATS. Using FEATS puts its PFA on the stack, then we add I to this to get the address of the byte. Then we fetch this byte, referred to as 'byte' in the table.

The key stage of the operation is the AND. This ANDs BYTE and byte to give 'BYTE . byte'. Let us see how this works. Suppose we have the following, in which the computer is testing to see if BYTE matches the byte for blackbird:

blackbird byte is	0 0 1 0 1 0 1 0
BYTE might be	0 0 0 1 1 0 1 0
AND gives	0 0 0 0 1 0 1 0

This value is *not* the same as BYTE. The bird described by BYTE has red feathers, so is not a blackbird. In another case the user might have seen a brown bird on the ground. The comparison would be:

blackbird byte	0 0 1 0 1 0 1 0
BYTE would be	0 0 1 0 1 0 0 0
AND gives	0 0 1 0 1 0 0 0

Now the value obtained by ANDing is *equal* to the value of BYTE. Although the user did not key in all the features of blackbird (i.e. the bird seen did not have black feathers since it was a female) all of the features keyed in are possessed by blackbirds. The bird *might* be a blackbird! The text relating to blackbirds is displayed. Looking at the table given earlier, it can be seen that the same result would be obtained with sparrow and with redbreast. All three names would be displayed.

If the ANDed value equals BYTE the IF action is executed. A line from Screen 30 (Fig. 12.2) is printed. The word 'dot-line' does this. It requires the stack to have the number of the Screen at top-of-stack and the line required at second-on-stack. The line required is I+1, since the screen lines are numbered from 1, while I is being incremented from zero.

The whole application is put together in the word BIRDS. '12 >VDU' clears the screen. Use the equivalent words in your version of FORTH. The first loop displays all eight lines of Screen 31 (Fig. 12.3), so displaying a numbered list of features. The user is then asked to key in the numbers. FEATURE accepts and processes these. The bits are gradually assembled in BYTE. FEATURES ends when zero is keyed. The computer then displays 'IT COULD BE:' followed by a list of birds conforming to the description, as obtained by GET.

If you are running FORTH on a disk system, the computer will automatically load the required text from disk. With a tape system, you may need to position the tape ready for loading Screens 30 and 31 during the course of the program. If your system allows three screens to be held in RAM at one time, then load all three screens, before running the application.

Here are two examples of BIRDS in action (see Fig. 12.5). This word works at very high speed making it possible for the computer to search through a hundred or more bytes in ten seconds or less. The

```
BIRDS
1  IS LESS THAN 20 CM LONG
2  HAS BLACK FEATHERS
3  HAS WHITE FEATHERS
4  HAS BROWN FEATHERS
5  HAS RED FEATHERS
6  SPENDS TIME ON THE GROUND
7  A WATER OR WATER-SIDE BIRD
8  NESTS IN TREES, NOT BUSHES
KEY THE NUMBERS:
4
8
0
IT COULD BE:
SPARROW, HSE: M. BLK. THROAT     64
OK


BIRDS
1  IS LESS THAN 20 CM LONG
2  HAS BLACK FEATHERS
3  HAS WHITE FEATHERS
4  HAS BROWN FEATHERS
5  HAS RED FEATHERS
6  SPENDS TIME ON THE GROUND
7  A WATER OR WATER-SIDE BIRD
8  NESTS IN TREES, NOT BUSHES
KEY THE NUMBERS:
1
2
0
IT COULD BE:
SPARROW, HSE: M. BLK. THROAT     64
MARTIN, HSE: WHT RUMP OF SWALLOW 56
SWALLOW: LONGER TAIL THAN MARTIN 54
TIT, GT: BLK STRIPE ON BELLY     36
OK
```

Fig. 12.5.

time taken depends more on the number of screens to be loaded and the number of names to be displayed, rather than on the number of bytes to be scanned.

To summarise

In this chapter you have found out how to:

- Handle bits within bytes.
- Store data in an array.
- Store text in screens.

You have used these FORTH words:

- AND performs bitwise ANDing on a 16-bit value (n1\n2 ... n1.n2).
- OR performs bitwise ORing on a 16-bit value (n1\n2 ... n1+n2). (NOTE '.' and '+' are the logical symbols for AND and OR).
- ('bracket' causes computer to ignore the material which follows it, until the delimiter,).
- CREATE used to create a word head in memory.
- C, 'C-comma' stores a value at the next available byte in the dictionary space (byte ...).
- 2* 'two-times' for fast multiplication by 2 (n ... 2*n).
- 2/ 'two-divide' for fast division by 2 (n ... n/2).
- .LINE 'dot-line' displays a line from a screen (line screen ...).

You have learned that:

- OR and AND are used for bit mapping and other logical operations.
- Bit-maps enable compact storage of data.
- Bit-maps are useful as the basis of identification applications.

Explore more

(1) Write a word SR to perform a right-shift on a binary number.

(2) Expand BIRDS to include more features and more birds.

(3) Write more words for BIRDS to:
   (a) improve the layout of the display.
   (b) check that the numbers typed in by the user are within the allowed range.

(c) display the message 'NONE FOUND' if no bird is found to match the description.

(4) Adapt BIRDS for use in other fields:

(a) Personnel or employee records: the computer picks out all male employees, for example, or those with special skills.

(b) Garden or house plants: the computer picks out plants suitable for growing in particular locations, or with flowers of a given colour, or which flower at a particular time of year.

(c) Cataloguing tape-recordings or records: the computer picks out recordings of pieces by a given composer, or of a specified type of music.

Appendix A
FORTH on Other Computers

This book was written using a version of FORTH (Acornsoft FORTH) which conforms to the 1979 Standard. As far as possible, only words which are likely to be found in other versions of FORTH have been used. Thus, there should be no difficulty in using this book with other versions. The only major exceptions are the special words used in Acornsoft FORTH which relate to the graphics features of the BBC Microcomputer and Electron. A few of these are used in Chapter Six. Your own version of FORTH should provide equivalent words for your own micro.

There are now so many versions of FORTH in existence for the popular microcomputers that it is not possible for us to be certain that every word used in this book is available in every version. Below we list some words which may not be available in your FORTH, with suggestions of how to implement them, using words that you are sure to have.

?DUP Duplicates the top-of-stack if it is not zero. This can be defined as:

```
: ?DUP DUP IF DUP THEN ;
```

In fig-FORTH this is called −DUP.

.S Displays the stack without altering it. This is the most difficult one to provide for in general terms, for it depends on the exact addresses used for the stack and for holding stack vectors in your system. Your handbook may suggest a definition if it is not already provided. The word in Fig. A.1 should work with most FORTHs. Before defining this version of 'dot-S', define an array VALS to hold as many values as you are ever likely to want displayed (see Chapter Six). Then define 'dot-S' as above. To use the word you must first place on the stack the number of values that you want to be displayed. The word takes the values off the stack in order, starting

```
: .S DUP 0 DO SWAP DUP . I VALS !
          LOOP 1 - -1 SWAP
          DO I VALS @ -1
          +LOOP ;
```

Fig. A1.

from top-of-stack, displaying them as it does so. It stores them in VALS. Then the second loop takes them from VALS, putting them back on the stack in their original order. Note that the stack is displayed with top-of-stack to the left.

BASE Some FORTHs use only a single byte for BASE. If this is so, the way to change the base is to use 'C-store', as in this example in which the computer is being put into binary mode:

```
2 BASE C!
```

CHARDEF This is not a regular FORTH word but one defined in Chapter Six for creating user-defined graphics. If your computer is capable of producing such graphics, your version of FORTH should include the necessary words to define CHARDEF. Consult the manual which came with your FORTH.

CMOVE Moves a block of bytes from one part of memory to another. As with the 'official' CMOVE, the version below does not check if the source block of memory and the destination block overlap. If they do, beware! It expects the stack to hold at:

top-of-stack	the number of bytes to be moved
second-on-stack	the first destination address
third-on-stack	the first source address.

If the number of bytes is 0, CMOVE takes no action except to remove all three values from the stack. The definition is shown in Fig. A.2.

```
: CMOVE DUP 0= IF DROP DROP
              ELSE OVER + ROT ROT DUP
                   ROT - ROT ROT
                   DO DUP I SWAP - C@
                      I C!
                   LOOP
              THEN DROP ;
```

Fig. A2.

COUNT This is described in Chapter Seven. Here is its definition:

```
: COUNT DUP C@ SWAP 1+ SWAP + ;
```

CREATE This may have a slightly different action in other versions of FORTH, or there may be another word having much the same action. In fig-FORTH the corresponding word is <BUILDS. The Jupiter Ace uses DEFINER. For example, the definition of the array-defining word (Chapter Six) is like this on the Ace:

```
DEFINER ARRAY 2 * ALLOT DOES> SWAP 2 * + ;
```

HEX Puts the computer into hexadecimal mode. This word is defined in one of the following ways, depending on whether the computer uses one byte or two for BASE:

```
: HEX 16 BASE C! ;
: HEX 16 BASE ! ;
```

KEY This waits for a key to be pressed, then places its ASCII code on the stack. **?KEY** is similar but waits for a limited period of time. The length of time it waits depends on the value of a number placed on the stack immediately before ?KEY is used. Some FORTHs have only INKEY, which accepts input from the keyboard, but does not wait. INKEY can be used in a definition of KEY:

```
: KEY BEGIN INKEY ?DUP UNTIL ;
```

Figure A3 is a definition of ?KEY based on INKEY. Placing 20000 on the stack makes the micro wait several tens of seconds, depending on the type of computer. If no key is pressed during the waiting time, ?KEY leaves zero on the stack.

```
: ?KEY BEGIN INKEY DUP IF SWAP DROP DUP
                INKEY  ? ELSE DROP 1 -
                         THEN
       UNTIL ;
```

Fig. A3.

NOT If this is not in your FORTH, use 0=, which has exactly the same action.

PLACEIT This is one of the special words defined in Chapter Six. If your FORTH has AT for displaying a character at a given row and column, PLACEIT may be defined like this:

```
: PLACEIT SWAP AT EMIT ;
```

On the Jupiter Ace, it is important to use the word INVIS to clear the upper part of the screen, ready for the graphics display.

WORD and **VARIABLE** have the same overall action in fig-FORTH, but differ in detail. Check their action in your version and make the necessary amendments when using these words. In some versions, WORD transfers text to a region of memory known as PAD, instead of to a special word buffer, but its action is otherwise the same. In some versions, VARIABLE expects an initial value (zero or any other number) on the stack when a variable is defined, instead of automatically setting the value to zero, as described in this book. Amend variable definitions to include this zero, for example:

```
0 VARIABLE RATE
```

Appendix B
ASCII Codes

Code	*Char.*	*Code*	*Char.*	*Code*	*Char.*
7	Bell or Beep	63	?	96	£
13	Carriage return	64	@	97	a
32	Space	65	A	98	b
33	!	66	B	99	c
34	"	67	C	100	d
35	#	68	D	101	e
36	$	69	E	102	f
37	%	70	F	103	g
38	&	71	G	104	h
39	'	72	H	105	i
40	(	73	I	106	j
41	)	74	J	107	k
42	*	75	K	108	l
43	+	76	L	109	m
44	,	77	M	110	n
45	−	78	N	111	o
46	.	79	O	112	p
47	/	80	P	113	q
48	0	81	Q	114	r
49	1	82	R	115	s
50	2	83	S	116	t
51	3	84	T	117	u
52	4	85	U	118	v
53	5	86	V	119	w
54	6	87	W	120	x
55	7	88	X	121	y
56	8	89	Y	122	z
57	9	90	Z	123	{
58	:	91	[	124	|
59	;	92	\	125	}
60	<	93	]	126	~
61	=	94	^		
62	>	95	_		

Index of FORTH Words

Including special words defined in the text

Subject Index

FORTH IS FAST, FORTH IS BRILLIANT – GO FORTH!

FORTH is a brilliant language with remarkable potential. It is four times as compact and up to ten times as fast as BASIC. It is also easy to learn. Interest in FORTH is growing rapidly and versions of FORTH are now available for an increasing range of micros.

This practical introduction to programming in FORTH has been written especially for beginners. Many examples are provided for you to try. Key them in and watch things happen on the screen. Try typing in something different and see if the effect is what you expect. This is the best, most creative way to learn FORTH fast.

If you want to be truly versatile, you cannot afford to ignore FORTH. This book shows you how to grasp this exciting new language and what you can achieve.

The Author

Owen Bishop is a freelance technical writer and programmer. He is the author of forty books including a number on popular computing. He is a well known and regular contributor to computing journals.

Cover photograph (background): Space Frontiers Limited

GRANADA PUBLISHING
Printed in Great Britain 0 246 12188 2

£6.95 net